

Hinweise zur Bearbeitung und Abgabe

Die Lösung der Hausaufgabe muss **eigenständig** erstellt werden. Abgaben, die identisch oder auffällig ähnlich zu anderen Abgaben sind, werden als **Plagiat** gewertet! **Plagiate sind Täuschungsversuche und führen zur Bewertung „nicht bestanden“ für die gesamte Modulprüfung.**

- Bitte nutzen Sie MARS zum Simulieren Ihrer Lösung. Stellen Sie sicher, dass Ihre Abgabe in MARS ausgeführt werden kann. Die Installation und das Ausführen von MARS wird im **5. Übungsblatt** beschrieben.
- Sie erhalten für jede Aufgabe eine separate Datei, die aus einem Vorgabe- und Lösungsabschnitt besteht. Ergänzen Sie bitte Ihren Namen und Ihre Matrikelnummer an der vorgegebenen Stelle. Bearbeiten Sie zur Lösung der Aufgabe nur den Lösungsteil unterhalb der Markierung:

```
#+ Loesungsabschnitt  
#+ -----
```

- Ihre Lösung muss auch mit anderen Eingabewerten als den vorgegebenen funktionieren. Um Ihren Code mit anderen Eingaben zu testen, können Sie die Beispieldaten im Lösungsteil verändern.
- Bitte nehmen Sie keine Modifikationen am Vorgabeabschnitt vor und lassen Sie die vorgegebenen Markierungen (Zeilen beginnend mit #+) unverändert.
- Eine korrekte Lösung muss die bekannten **Registerkonventionen** einhalten. Häufig können trotz nicht eingehaltener Registerkonventionen korrekte Ergebnisse geliefert werden. In diesem Fall werden trotzdem Punkte abgezogen.
- Falls Sie in Ihrer Lösung zusätzliche Speicherbereiche für Daten nutzen möchten, verwenden Sie dafür bitte ausschließlich den **Stack** und keine statischen Daten in den Datensektionen (.data). Die Nutzung des Stacks ist gegebenenfalls notwendig, um die Registerkonventionen einzuhalten.
- Die zu implementierenden Funktionen müssen als Eingaben die Werte in den **Argument-Registern** (\$a0–\$a3) nutzen. Daten in den Datensektionen der Assemblerdatei dürfen **nicht** direkt mit deren Labels referenziert werden.
- Bitte gestalten Sie Ihren Assemblercode nachvollziehbar und verwenden Sie detaillierte Kommentare, um die Funktionsweise Ihres Assemblercodes darzulegen.
- Die Abgabe erfolgt über ISIS. Laden Sie die zwei Abgabedateien separat hoch.

Aufgabe 1: Caesar-Verschlüsselung (10 Punkte)

Aufgabe: Bei der Caesar-Verschlüsselung¹ handelt es sich um ein einfaches Verschlüsselungsverfahren, wodurch Texte unkenntlich gemacht werden. Dabei werden die einzelnen Zeichen im zu verschlüsselnden Text durch zyklisch verschobene Zeichen ersetzt.

Implementieren Sie die Funktion `caesar`, welche die zu verschlüsselnde Zeichenkette `text` und einen Wert `shift` für die zyklische Verschiebung übergeben bekommt und die Zeichenkette `text` gemäß der Caesar-Verschlüsselung modifiziert. Die C-Signatur der zu implementierenden Funktion lautet:

```
void caesar( char* text, int shift );  
           $a0      $a1
```

¹Weitere Hintergrundinformationen: <https://de.wikipedia.org/wiki/Caesar-Verschlüsselung>

Bei `text` handelt es sich um einen Pointer zum ersten Zeichen einer Zeichenkette. Diese Zeichenkette endet mit einem Nullterminator. Jedes Zeichen ist ein Byte groß und kann anhand des ASCII-Zahlenwerts untersucht werden. Eine ASCII-Tabelle finden Sie zum Beispiel auf der MIPS Green Card.

Verschlüsselung: Die Verschlüsselung erfolgt durch zyklische Verschiebung der Zeichen um den Wert `shift`. Dabei werden nur alphanumerische Zeichen (Groß- und Kleinbuchstaben sowie Ziffern) verschoben. Alle anderen Zeichen bleiben unverändert. Die Verschiebung erfolgt innerhalb der jeweiligen Zeichengruppe (Großbuchstaben, Kleinbuchstaben, Ziffern) und ist zyklisch, das heißt, dass nach dem letzten Zeichen der Gruppe wieder zum ersten Zeichen zurückgekehrt wird. Die Zeichengruppen sind wie folgt definiert:

- Großbuchstaben: 'A' bis 'Z'
- Kleinbuchstaben: 'a' bis 'z'
- Ziffern: '0' bis '9'

Beispiel: Mit dem folgenden Wert für `text` und einer Verschiebung `shift` von 5 wird die Vorgehensweise der Verschlüsselung veranschaulicht:

`text = "Bsp Txt1!"`

Jedes alphanumerische Zeichen wird in der jeweiligen Zeichengruppe schrittweise um eine Position verschoben. Sobald das Ende einer Zeichengruppe erreicht ist, beginnt die Vergabe der Zeichen wieder von vorne (↓). Nach 5 Schritten ergibt sich folgendes Bild:

Originaltext:	B	s	p	T	x	t	1	!
shift = 1	C	t	q	U	y	u	2	!
shift = 2	D	u	r	V	z	v	3	!
shift = 3	E	v	s	W	a	w	4	!
shift = 4	F	w	t	X	b	x	5	!
shift = 5	G	x	u	Y	c	y	6	!

Test-Eingaben: Testen Sie Ihre Lösung mit unterschiedlichen Eingaben! Bearbeiten Sie dafür die Zeichenkette `test_text` und die Verschiebungsvariable `test_shift`. Folgende Tabelle enthält Beispieleingaben und die erwarteten Ergebnisse, welche zum Testen verwendet werden können:

Eingabe text	Verschiebung shift	Erwarteter Text in text
Bsp Txt1!	5	Gxu Ycy6!
Hello, World!	10	Rovvy, Gybn!
MIPS Assembly 2025	21	HDKN Vnnzhwgt 3136
a1_caesar	1	b2_dbftbs
Lz3k o6k 5fujqhl5v	8	Th1s w4s 3ncrypt3d

Hinweise:

- Beachten Sie, dass die Eingabe-Zeichenkette `text` beliebig lang oder kurz sein kann, aber stets mit einem Nullterminator beendet wird.
- Sie können davon ausgehen, dass die Verschiebung `shift` im Bereich von 0 bis 25 liegt.
- Um eine Modulo-Operation ($\$t0 = \$t1 \bmod \$t2$) durchzuführen, können Sie den MIPS-Befehl `div $t1, $t2` verwenden und anschließend den Restwert mit `mfhi $t0` auslesen.

Aufgabe 2: Bloom-Filter auswerten (10 Punkte)

Aufgabe: Bei einem Bloom-Filter² handelt es sich um eine probabilistische Datenstruktur, mit welcher schnell geprüft werden kann, ob eine gegebene Zeichenkette in einer Erkennungsmenge enthalten ist. Dabei kann es zu falsch positiven Ergebnissen, jedoch nicht zu falsch negativen Ergebnissen kommen. Die Basis unseres Bloom-Filters ist eine zweidimensionale Bitmatrix der Größe 5×32 . Jedes Element in der Erkennungsmenge wird dabei durch eine bestimmte Anzahl an gesetzten Bits in der Bitmatrix kodiert. Die genauen Positionen der Bits werden dabei durch eine Hashfunktion bestimmt. In dieser Aufgabe soll die Funktion `bloom_evaluate` implementiert werden. Das Argument `text` stellt eine zu prüfende Zeichenkette dar, welche gegenüber des in der `bitmatrix` kodierten Bloom-Filters ausgewertet werden soll. Die Funktion soll prüfen, ob die Zeichenkette `text` möglicherweise in der Erkennungsmenge des Bloom-Filters enthalten ist. Das Argument `amt` gibt dabei die Anzahl der Hashfunktionsaufrufe an. Die C-Signatur der zu implementierenden Funktion lautet:

<code>int</code>	<code>bloom_evaluate(</code>	<code>char* text,</code>	<code>int* bitmatrix,</code>	<code>int amt</code>	<code>);</code>
<code>\$v0</code>		<code>\$a0</code>	<code>\$a1</code>	<code>\$a2</code>	

Bei `text` handelt es sich um einen Pointer zum ersten Zeichen einer Zeichenkette. Diese Zeichenkette endet mit einem Nullterminator. Jedes Zeichen ist ein Byte groß.

Das Argument `bitmatrix` stellt einen Pointer auf die Bitmatrix des Bloom-Filters dar. Die Bitmatrix ist dabei als eindimensionales Array von 5 `int`-Werten kodiert, wobei jeder `int`-Wert 32 Bits enthält. Das erste Element des Arrays repräsentiert dabei die erste Zeile der Bitmatrix, das zweite Element die zweite Zeile und so weiter.

Hilfsfunktion: Die Hilfsfunktion `hash` implementiert eine Hashfunktion. Dabei werden die Zeichenkette `text` mit der Länge `len` und einem Startwert `seed` als Argumente übergeben. Die Funktion liefert einen Hashwert als Rückgabewert. Die C-Signatur der Hilfsfunktion lautet:

<code>int</code>	<code>hash(</code>	<code>char* text,</code>	<code>int len,</code>	<code>int seed</code>	<code>);</code>
<code>\$v0</code>		<code>\$a0</code>	<code>\$a1</code>	<code>\$a2</code>	

²Weitere Hintergrundinformationen: <https://de.wikipedia.org/wiki/Bloomfilter>

Vorgehen: Für die Eingaben-Zeichenkette `text` soll überprüft werden, ob diese in der Erkennungsmenge des Bloom-Filters enthalten ist. Die Hilfsfunktion `hash_muss` dabei als Hashfunktion aufgerufen werden. Die Vorgehensweise wird im Folgenden erläutert:

1. *Zählen:* Bestimme die Länge der Zeichenkette `text`. Die Zeichenkette wird beendet durch einen Null-terminator, welcher nicht mitgezählt werden soll.
2. *Hashing:* Rufe die Hashfunktion `amt`-mal auf, um `amt` verschiedene Hashwerte zu erhalten. Bei dem ersten Aufruf soll als seed 0 übergeben werden. Bei jedem weiteren Aufruf soll der seed jeweils um 1 erhöht werden.
3. *Bitpositionen bestimmen:* Für jeden erhaltenen Hashwert soll die zugehörige Bitposition in der Bitmatrix bestimmt werden:
 - Der Zeilenindex `row` der Bitmatrix wird durch den Hashwert modulo 5 bestimmt: $row = hash_value \bmod 5$.
 - Der Spaltenindex `col` der Bitmatrix wird durch den Hashwert modulo 32 bestimmt: $col = hash_value \bmod 32$.
4. *Bits prüfen:* Prüfe, ob die Bits an den jeweiligen Position (`row`, `col`) in der Bitmatrix gesetzt sind. Falls eines der geprüften Bits nicht gesetzt ist, so ist die Zeichenkette `text` definitiv nicht in der Erkennungsmenge des Bloom-Filters enthalten. In diesem Fall soll die Funktion den Wert 0 zurückgeben. Falls alle geprüften Bits gesetzt sind, so ist die Zeichenkette `text` möglicherweise in der Erkennungsmenge des Bloom-Filters enthalten. In diesem Fall soll die Funktion den Wert 1 zurückgeben.

Test-Eingaben: Testen Sie Ihre Lösung mit unterschiedlichen Eingaben! Bearbeiten Sie dafür die Zeichenkette `test_text`, die Bitmatrix `test_bitmatrix` und die Anzahl der Hashfunktionsaufrufe `test_amt`. Folgende Tabelle enthält Beispielingaben und die erwarteten Ergebnisse, welche zum Testen verwendet werden können:

Eingabe text	bitmatrix	amt	Erwarteter Rückgabewert
Apfel	A	4	1
MIPS	A	4	0
Orange	A	4	1
Birne	A	4	1
Birne	A	5	0
Kohlrabi	B	5	1
Apfel	B	5	0
Zwiebel	B	5	1
Zwiebel	B	3	1
Gurke	B	5	1

$$A = \begin{pmatrix} 0x10000020 \\ 0x04009440 \\ 0x02064224 \\ 0x08000240 \\ 0x02008420 \end{pmatrix}$$

$$B = \begin{pmatrix} 0x05004400 \\ 0x31400040 \\ 0x22310020 \\ 0x10509510 \\ 0x20104110 \end{pmatrix}$$

Hinweise:

- Die Eingabe-Zeichenkette `text` kann beliebig lang oder kurz sein, endet aber stets mit einem Nullterminator.
- Um eine Schiebe-Operation ($\$t0 = \$t1 \ll \$t2$) um einen variablen Faktor `$t2` durchzuführen, können Sie den MIPS-Befehl `sllv $t0, $t1, $t2` verwenden.
- Um eine Modulo-Operation ($\$t0 = \$t1 \bmod \$t2$) durchzuführen, können Sie den MIPS-Befehl `div $t1, $t2` verwenden und anschließend den Restwert mit `mfhi $t0` auslesen.