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.
- 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: Kapitalisierung zählen (10 Punkte)

Aufgabe: Implementieren Sie die Funktion `cap_words`, welche die Anzahl von groß geschriebenen Wörtern in einem String zurückgibt. Die C-Signatur der zu implementierenden Funktion ist:

`int cap_words( char* string);`
`$v0` `$a0`

Bei `string` 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.

Wörter sollen dann als groß gewertet werden, wenn es sich beim ersten Buchstaben des Worts um einen Großbuchstaben handelt. Wörter sind immer durch Leerzeichen getrennt. Großbuchstaben, die nicht als erster Buchstabe eines Worts vorkommen, sollen nicht gezählt werden. Zusätzlich zu Buchstaben

und Leerzeichen können Satzzeichen und Sonderzeichen vorkommen, welche zu ignorieren sind. Umlaute kommen nicht vor.

Beispiele: Die Vorgabe enthält zum Testen den String `test_string`, den Sie gerne editieren können, um Ihre Lösung mit weiteren Beispielen zu kontrollieren. Folgende Tabelle enthält Beispieleingaben und die erwarteten Rückgabewerte, welche zum Testen verwendet werden können:

Eingabe-String	**Erwarteter Rückgabewert**
„hier sind nur Substantive gross: Affe, Banane, Clown, denken“	4
„eins 1, Zwei 2, Drei 3, A, B, C“	5
„Kurz Aber VIELE as: Aa A Aa A AaaaAa“	8
„eIn bUchstAbE GrOss rEIcht nIcht;“	1
„Nur aNfangsbuchstaben ZAEhlen als gROss geschrieben.“	2

Aufgabe 2: Permutationen durchlaufen (10 Punkte)

Hintergrund: Eine Permutation („Vertauschung“) ist eine Anweisung, wie Elemente eines Arrays umgeordnet werden sollen. Es gibt $n!$ Möglichkeiten, n Elemente anzuordnen[1]. Daher gibt es $n!$ verschiedene Permutationen der Länge n. Bei manchen Anwendungen ist es hilfreich, alle Permutationen einer bestimmten Länge in einer bestimmten Reihenfolge zu durchlaufen. Permutationen können beispielsweise in lexikographischer Reihenfolge durchlaufen werden, das heißt in der Reihenfolge, in der diese in einem Wörterbuch sortiert aufgelistet wären[2].

Aufgabe: Implementieren Sie die Funktion `nextperm`. Diese Funktion soll den lexikographischen Nachfolger der übergebenen Permutation `perm` der Länge `length` ermitteln. `perm` soll durch eine Reihe von Aufrufen der vorgegebenen Hilfsfunktion `swap` in den lexikographischen Nachfolger umgewandelt werden (*in-place*). Die vorgegebene Hilfsfunktion `swap` muss verwendet werden. Bei Erfolg soll `nextperm` den Wert `1` zurückgeben. Falls `perm` die letzte Permutation der übergebenen Länge ist, soll `nextperm` den Wert `0` zurückgeben. Die C-Signatur der zu implementierenden Funktion lautet:

```
int  nextperm(  char *perm,  int length);
```

`$v0`	`$a0`	`$a1`

Beispiel: Tabelle 1 zeigt die Permutationen der Länge 4 in lexikographischer Ordnung. Dies ist die erwartete Ausgabe des Testprogramms für die vorgegebene Startpermutation $(0 \quad 1 \quad 2 \quad 3)$. Sie können Ihren Programmcode auch für andere Startpermutationen testen, indem Sie die Werte `test_perm` und `test_perm_length` in der Vorgabedatei anpassen.

Das Testprogramm bricht ab, sobald `nextperm` 0 zurückgibt, oder nachdem `nextperm` 1000-mal aufgerufen wurde.

Tabelle 1: Die Permutationen der Länge 4 in lexikographischer Reihenfolge.

Iteration	Permutation			
(0):	0	1	2	3
(1):	0	1	3	2
(2):	0	2	1	3
(3):	0	2	3	1
(4):	0	3	1	2
(5):	0	3	2	1
(6):	1	0	2	3
(7):	1	0	3	2
(8):	1	2	0	3
(9):	1	2	3	0
(10):	1	3	0	2
(11):	1	3	2	0
(12):	2	0	1	3
(13):	2	0	3	1
(14):	2	1	0	3
(15):	2	1	3	0
(16):	2	3	0	1
(17):	2	3	1	0
(18):	3	0	1	2
(19):	3	0	2	1
(20):	3	1	0	2
(21):	3	1	2	0
(22):	3	2	0	1
(23):	3	2	1	0

[1] $n!$ ist Fakultät von n, also das Produkt aller natürlichen Zahlen ≥ 1 und $\leq n$.

[2] Die lexikographische Reihenfolge kann mathematisch definiert werden als: Permutation p steht lexikographisch vor Permutation q genau dann wenn es ein k gibt, für das gilt $p(k) < q(k)$ und $p(i) = q(i)$ für alle $i < k$.

Hilfsfunktion: `swap` tauscht zwei Elemente anhand der übergebenen Indizes i und j. Als Parameter `perm` soll der Pointer auf das erste Element des zu permutierenden Arrays übergeben werden. Die C-Signatur lautet:

`void swap(`	`char *perm,`	`int i,`	`int j);`
	`$a0`	`$a1`	`$a2`

Algorithmus[3]. Der Algorithmus zur Bestimmung des lexikographischen Nachfolgers einer Permutation besteht aus vier Schritten:

1. Zunächst sucht man den größten Index k für den gilt, dass das Element des Arrays an der Stelle k kleiner als jenes an der Stelle $k+1$ ist, also $\texttt{perm}[k] < \texttt{perm}[k+1]$. Gibt es keinen solchen Index, endet der Algorithmus (Rückgabewert 0).
2. Der zweite Schritt findet den größten Index l, wobei $l > k$ und dabei auch $\texttt{perm}[l] > \texttt{perm}[k]$ sein muss.
3. Der dritte Schritt besteht aus dem Vertauschen der Elemente an den Indizes k und l. Dies muss durch den Aufruf der Funktion `swap` erfolgen.
4. Als vierter und letzter Schritt wird der dem Element k nachfolgende Teil des Arrays, also $k+1$ bis zur Länge `length` des Arrays, in umgekehrter Reihenfolge angeordnet. Nutzen Sie dafür erneut die Funktion `swap`, diesmal in einer Schleife. Es können beispielsweise zwei Indizes i und j genutzt werden, wobei in jedem Schritt i inkrementiert und j dekrementiert wird bis i gleich oder größer als j ist.

Hinweise:

- Anders als in einer früheren Hausaufgabe wird die Permutation in dieser Aufgabe in einem Byte-Array (`char[]` / `char *`) gespeichert.
- Jede gültige Permutation der Länge n enthält jede ganze Zahl zwischen 0 und $n-1$ genau einmal.
- Sie können davon ausgehen, dass die Länge der Permutation zwischen 1 und 127 liegt. Es sind nur positive Zahlen im Array enthalten.
- Funktionsweise des Algorithmus: Betrachtet man eine Permutation als Folge von n verschiedenen Ziffern von 0 bis $n-1$, d.h. jedes Element der Permutation als eine Ziffer im Zahlensystem zur Basis n, dann ist die nächste Permutation die erste nachfolgende natürliche Zahl, die auch n verschiedene Ziffern hat. Anders als bei einer natürlichen Zahl dürfen in einer Permutation Ziffern nicht mehrmals vorkommen. Der nächste Wert einer Ziffer überspringt alle Ziffernwerte, die links davon schon vorkommen und nimmt den nächstgrößeren Ziffernwert weiter rechts an. Ein Übertrag entsteht, wenn alle größeren Ziffernwerte nicht mehr verfügbar sind, also alle Ziffernwerte weiter rechts kleiner sind. Wie bei gewöhnlicher Addition wird nach einem Übertrag die nächste Ziffer links auch erhöht. Nach einem Überlauf kann der Wert einer Ziffer erst bestimmt werden, wenn alle Ziffern links davon bestimmt wurden.

  Um die nächste Permutation analog zu einer Zahl zu berechnen, ohne jede Nachfolgezahl zu überprüfen, sucht der Algorithmus zuerst die letzte Ziffer, die bei Erhöhung keinen weiteren Übertrag erzeugt. Das ist die letzte Ziffer von links nach rechts, die eine größere Ziffer weiter rechts von sich hat. Nachdem die gefundene Ziffer durch Tauschen mit der kleinsten größeren Ziffer weiter rechts erhöht wurde, nehmen alle Ziffernpositionen mit Übertrag jeweils von links nach rechts den nächsten noch freien Wert an, was zu einer aufsteigenden Ziffernfolge und somit zu einer Umkehrung der zuvor absteigenden Ziffernfolge führt.

[3] Der beschriebene Algorithmus ist ebenfalls auf Wikipedia zu finden:
https://en.wikipedia.org/wiki/Permutation#Generation_in_lexicographic_order