Aufgabensammlung ADS-Repetitorium 2021

Rekursive Laufzeit – Datenstrukturen

Aufgabe 1: Rekursive Laufzeiten

Finden Sie für die nachstehenden Rekursionsgleichungen jeweils eine Funktion f, für die $T \in \Theta(f)$ gilt. Sie können davon ausgehen, dass die Laufzeit im Basisfall konstant ist.

(a) $T(n) = 4T(\lfloor n/2 \rfloor) + \frac{1}{2}n^2\sqrt{n}$

Lösung: $T(n) \in \Theta(0,5n^2\sqrt{n})$.

(b) $T(n) = 4T(\lfloor n/2 \rfloor) + n^2 \log n + n$

Lösung: $T(n) \in \Theta(n^2 \log^2 n)$.

(c) $T(n) = T(n-3) + 2n$

Lösung: $T(n) \in \Theta(n^2)$.

(d) $T(n) = 2T(\lfloor n/4 \rfloor) + 3\sqrt{n}$

Lösung: $T(n) \in \Theta(\sqrt{n} \log n)$.

(e) $T(n) = 3T(\lfloor n/2 \rfloor) + \frac{n}{6}$

Lösung: $T(n) \in \Theta(n^{\log_2 3})$.

(f) $T(n) = 3T(\lfloor n/5 \rfloor) + \frac{1}{2}\sqrt{n}$

Lösung: $T(n) \in \Theta(n^{\log_5 3})$.

(g) $T(n) = 12T(\lfloor n/2 \rfloor) + n^4$

Lösung: $T(n) \in \Theta(n^4)$.

(h) $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \mathcal{O}(n \log n)$

Lösung: $T(n) \in \Theta(n \log^2 n)$.

Aufgabe 2: Rekursive Algorithmen

Geben Sie jeweils einen Algorithmus an, der die gegebene Laufzeit erfüllt. Alle Algorithmen sollen im Basisfall, also für $n = 1$, konstante Laufzeit haben. Als Eingabe bekommen die Algorithmen jeweils ein Feld der Länge n. Die Algorithmen sollten eine Rückgabe haben, doch der Wert der Rückgabe ist egal. Sie dürfen Rundungsfehler ignorieren.

(a) $T(n) = 3T(n-1) + n$

Lösung:

Algorithmus 1: recursiveAlgo(A[])

```
1 if A.size ≤ 1 then
2     return 42
3 for i = 1 to n do
4     A[i] = i
5 recursiveAlgo(A[1..n − 1])
6 recursiveAlgo(A[1..n − 1])
7 recursiveAlgo(A[1..n − 1])
8 return A[1]
```

(b) $T(n) = 4T(n/4) + O(n)$

Lösung:

Algorithmus 2: recursiveAlgo(A[])

```
1 if A.size ≤ 1 then
2     return 42
3 for i = 1 to n do
4     A[i] = i
5 recursiveAlgo(A[1..n/4])
6 recursiveAlgo(A[1..n/4])
7 recursiveAlgo(A[1..n/4])
8 recursiveAlgo(A[1..n/4])
9 return A[1]
```

(c) $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + T(\sqrt{n})$; bitte beachten Sie hier die Rundungen.

Lösung:

Algorithmus 3: recursiveAlgo(A[])

```
1 if A.size ≤ 1 then
2     return 42
3 for i = 1 to √n do
4     A[i] = i
5 recursiveAlgo(A[1..⌈n/2⌉])
6 recursiveAlgo(A[1..⌊n/2⌋])
7 return A[1]
```

(d) $T(n) = 3T(n-5) + n \log n$

Lösung:

Algorithmus 4: recursiveAlgo(A[])

```
1 if A.size ≤ 1 then
2     return 42
3 MergeSort(A)
4 recursiveAlgo(A[1..n − 5])
5 recursiveAlgo(A[1..n − 5])
6 recursiveAlgo(A[1..n − 5])
7 return A[1]
```

(e) $T(n) = T(n-1) + T(n-2) + 42T(n/2) + n^2$

Lösung:

Algorithmus 5: recursiveAlgo(A[])

```
1 if A.size ≤ 1 then
2     return 42
3 InsertionSort(A)
4 recursiveAlgo(A[1..n − 1])
5 recursiveAlgo(A[1..n − 2])
6 for i = 1 to 42 do
7     recursiveAlgo(A[1..n/2])
8 return A[1]
```

Aufgabe 3: Rekursive Gleichung aufstellen
Gegeben sei folgender Algorithmus:

Algorithmus 6: RecursiveAlgo(**int** A[ ], l= 1, r=A.length)

```
1 if l < r then
2     m = ⌊(l + r)/2⌋
3     RecursiveAlgo(A, l, m)
4     RecursiveAlgo(A, m + 1, r)
5     InsertionSort(A, l, r)
```

(a) Stellen Sie eine Rekursionsgleichung T für den gegebenen Algorithmus auf.

Lösung:

$$T(n) = \begin{cases} 2 \cdot T(n/2) + n^2 & \text{falls } n > 1 \\ 1 & \text{sonst} \end{cases}$$

(b) Finden Sie eine Funktion f, für die $T \in \Theta(f)$ gilt.

Lösung: $T(n) \in \Theta(n^2)$.

Aufgabe 4: Löschen in einer Hash-Tabelle

Gegeben sei eine Hashtabelle H mit einer Hash-Funktion $h(x, i)$. Es wird offene Adressierung verwendet.

(a) Beschreiben Sie in Worten, wie der Algorithmus Search(**int** k) aus der Vorlesung funktioniert.

Lösung: Der Algorithmus bildet den Hash h von k und prüft H[h]. Wenn der Inhalt dem gesuchten Wert entspricht, wird dieser zurückgegeben, ansonsten wird i inkrementiert und mit der verwendeten Methode zur Auflösung von Kollisionen ein neuer Hash h gebildet. Das wird wiederholt, bis entweder ein leerer Eintrag oder der gesuchte Wert gefunden wird.

(b) Ein Element k soll aus der Tabelle gelöscht werden. Warum sollte man den Wert nicht mit der folgenden Befehlsfolge löschen?
$j =$ Search(k)
H$[j] = -1$

Lösung: Nach dem in a) beschrieben Vorgehen der Methode Search(k) bricht die Suche ab, wenn ein leerer Eintrag gefunden wird. Wenn nun mitten in der Sondierfolge ein Eintrag gelöscht wird, bricht die Suche nach einem anderen, später eingefügten Element eventuell zu früh ab und findet das Element nicht mehr.

(c) Implementieren Sie die Operation Delete(**int** k), die einen Schlüssel aus der Tabelle löscht, ohne dass das Problem aus b) auftritt. *Tipp*: Verwenden Sie einen besonderen Wert, um gelöschte Zellen zu markieren.

Lösung:

Algorithmus 7: Delete(**int** k)

```
1 j = Search(k)
2 H[j] = deleted value
```

(d) Welche Änderung muss nun in den Methoden Insert(**int** k) und Search(**int** k) vorgenommen werden?

Lösung: Die Insert(**int** k)-Methode muss nun jeden Wert, der ihr während der Sondierreihenfolge begegnet, auf den *special value* überprüfen. Wenn sie einen solchen findet, darf sie ihn ersetzen. In der Search(**int** k) muss keine Änderung vorgenommen werden, da der *special value* ungleich dem leeren Wert ist. Die Suche geht also einfach über die gelöschte Zelle hinweg.

(e) Beschreiben Sie kurz die Auswirkungen Ihrer Änderungen auf die Laufzeit der Operationen.

Lösung: Die Laufzeit der Insert(**int** k) ändert sich nicht, aber die Laufzeit von Search(**int** k) verschlechtert sich, wenn das gesuchte Element nicht in der Datenstruktur vorhanden ist. Nehmen wir an, alle Werte aus einer ehemals vollen Datenstruktur wurden gelöscht. Die Search(**int** k)-Funktion muss nun trotzdem alle Felder betrachten, da in allen Feldern der *special value* steht, bevor sie **false** zurückgibt.

Aufgabe 5: Doppeltes Hashing

Welche der folgenden Funktionen eignen sich für eine Hashtabelle der Länge 25, wenn doppeltes Hashing verwendet wird und die Hashfunktion $h(k, i) = (h_0(k) + ih_1(k)) \bmod 25$ mit $h_0(k) = (4k + 2) \bmod 25$ ist? Begründen Sie Ihre Entscheidungen.

(a) $h_1(k) = 1$

Lösung: Geeignet.

(b) $h_1(k) = 9 - (k \bmod 4)$

Lösung: Geeignet.

(c) $h_1(k) = k \bmod 17$

Lösung: Ungeeignet.

(d) $h_1(k) = (3 + 5k) \bmod 25$

Lösung: Geeignet.

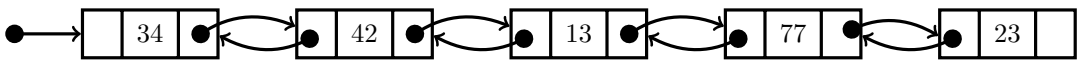
(e) $h_1(k) = (4k - 1) \bmod 13$

Lösung: Ungeeignet.

Aufgabe 6: Doppelt-Verkettete Listen
Gegeben sei folgender Algorithmus

Algorithmus 8: modifyList(List L)

```
1  item = L.head
2  size = 1
3  while item.next != null do
4  |   item = item.next
5  |   size = size + 1
6  item = L.head
7  for i = 1 to ⌊size/2⌋ do
8  |   item = item.next
9  |   item.prev.next = item.next
10 |   item.next.prev = item.prev
11 |   item = item.next
```

(a) Sei $L =$ 

.

Zeichnen Sie die Liste für jede Iteration der Schleife in Zeile 7.

(b) Beschreiben Sie, was der Algorithmus allgemein macht.

Lösung: Der Algorithmus löscht jedes zweite Element, angefangen beim zweiten, aus der Liste.

(c) Der Algorithmus enthält zwei Fehler. Geben Sie eine Liste an, sodass die Ausführung von Zeile 3 fehlschlägt. Geben Sie außerdem eine Liste an, die zu einem Fehler in Zeile 10 führt. Verbessern Sie den Pseudocode.

Lösung: Falls die Liste leer ist, also L.head ist leer, dann wird ein Fehler in Zeile drei erzeugt. Zur Verbesserung muss man zu Beginn des Algorithmus eine Abfrage durchführen, ob die Liste leer ist. Ist dies der Fall, kann man die Ausführung des Algorithmus sofort abbrechen. Falls die

Liste eine gerade Länge hat, wird versucht, das letzte Element zu löschen. item.next ist aber nicht definiert, sodass auf item.next.prev nicht zugriffen werden kann. Man kann dies ebenfalls durch eine entsprechende Abfrage lösen.

(d) Was würde passieren, wenn Zeile 11 gelöscht würde?

Lösung: Es wird die Subliste angefangen bei Element 2 bis zum ersten Element nach der Hälfte (inklusive) gelöscht.

(e) Welche Augmentierung der Datenstruktur List schlagen Sie vor, um den Code zu verkürzen?

Lösung: Durch die Verwendung eines Attributs size, welches die Länge der Liste angibt, können die ersten 5 Zeilen gelöscht werden.

Aufgabe 7: Ringe

Ein Ring ist eine Datenstruktur, die auf einer doppelt-verketteten Liste aufbaut. Der Unterschied zwischen beiden Datenstrukturen ist, dass beim Ring die Attribute next und prev niemals nil sind und über next eines beliebigen Elements jedes andere Element erreicht werden kann (analog auch über prev in die andere Richtung). Jeder Ring hat einen Pointer entry auf einen beliebiges Element im Ring. Ansonsten gibt es die gleichen Operationen wie bei der Liste, wobei insert(k) vor dem aktuellen entry einfügt und dann entry aufs neue Item setzt.

(a) Zeichnen Sie den Ring, der die ersten vier Fibonacci-Zahlen enthält. Der Pointer entry soll auf eine gerade Primzahl zeigen.

Lösung: $1 \leftrightharpoons 1 \leftrightharpoons 2 \leftrightharpoons 3 \leftrightharpoons 5$, sowie zwischen 5 und 1 besteht eine wechselseitige Verbindung. Der Pointer entry zeigt auf die 2.

(b) Implementieren Sie die Methode makeRing(List l), die aus einer doppelt-verketteten Liste einen Ring macht. Der Pointer entry des entstanden Rings soll dabei auf den Kopf der ursprünglichen Liste zeigen. Die Liste darf verändert werden.

Lösung:

Algorithmus 9: makeRing(List list)

```
1 tail = list.head
2 if tail = null then return
3 while tail.next ≠ null do
4 |  tail = tail.next
5 list.head.prev = tail
6 tail.next = list.head
7 ring = new Ring()
8 ring.entry = list.head
9 return ring
```

(c) Implementieren Sie die Methode split(Ring r, Item i, Item j), die den Ring r in zwei Ringe aufspaltet. Dabei sollen alle Items zwischen i und j (inklusive, in Richtung des next-Attributs) aus r gelöscht werden und als eigener Ring zurückgegeben werden. Weisen Sie die entry-Werte beliebig, aber gültig, zu. Gehen Sie davon aus, dass mindestens ein Element in r verbleibt (mit anderen Worten i.prev $\neq$ j).

Lösung:

Algorithmus 10: split(Ring r, Item i, Item j)

```
1 ring.entry = j.next
2 i.prev.next = j.next
3 j.next.prev = i.prev
4 i.prev = j
5 j.next = i
6 extracted = new Ring()
7 extracted.entry = j
8 return extracted
```

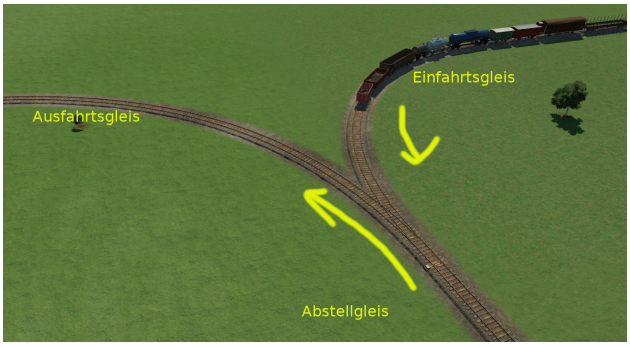
(d) Implementieren Sie die Methode merge(Ring r, Ring u), die die Items des Rings u vor r.entry einfügt. Achten Sie darauf, dass die Reihenfolge der Elemente innerhalb der Ringe gleich bleibt.

Lösung:

Algorithmus 11: merge(Ring r, Ring u)

```
1 r.entry.prev.next = u.entry.prev
2 u.entry.next.prev = r.entry.prev
3 r.entry.prev = u.entry
4 u.entry.next = r.entry
```

Aufgabe 8: Waggons stapeln



Wir betrachten einen Zug mit n verschiedenen Güterwagons, die mit den Zahlen 1 bis n aufsteigend beschriftet sind. Wir betrachten folgenden Algorithmus:

1. Nimm vom Anfang des Zuges eine zufällige Anzahl von Waggons.
2. Schiebe diese Waggons auf das Abstellgleis.
3. Nimm eine zufällige Anzahl von Waggons auf dem Abstellgleis und schiebe sie aufs Ausfahrtsgleis.
4. Wiederhole ab 1., bis alle Waggons auf dem Ausfahrtgleis sind.

(a) Formulieren Sie den obigen Algorithmus in Pseudocode. Die Eingabe ist eine Zahl n, die Ausgabe soll ein entsprechend permutiertes Feld der Zahlen 1 bis n sein.

Lösung: Idee: Fasse Abstellgleis als Stapel auf und verwalte Waggons in doppelt-verketteter Liste, damit wir einfacher löschen können.

(b) Geben Sie einen Algorithmus an, der für eine Waggonfolge entscheidet, ob diese durch dieses Verfahren zustande gekommen ist. Beispiel: Die Folge 3 – 2 – 4 – 1 ist entstanden, indem zuerst die Wagons 1, 2 und 3 auf das Abstellgleis wanderten. Dann wurde Waggons 3 und 2 aufs Ausfahrtsgleis gestellt, danach Waggon 4 vom Einfahrtsgleis aufs das Ausfahrtsgleis umgeparkt und zuletzt Waggon 1 hinten an den Zug angehängt.

Lösung:
Die Idee ist es, den Rangiervorgang rückgängig zu machen. Dazu gehen wir das Eingabefeld rückwärts durch. Solange die Zahlen dabei auf dem Weg nach vorne steigen, speichern wie sie in einem Stapel. Dies funktioniert, da wir die kleineren Zahlen praktisch zwischenspeichern müssen. Sobald auf dem Weg von hinten die Zahlen jedoch wieder kleiner werden, nehmen wir die großen Zahlen vom Stapel und fügen sie in die Ausgabeliste ein. Wenn diese am Ende die Zahlen in korrekter Reihenfolge enthält, dann war die Eingabefolge gültig.

Aufgabe 9: Henne-Ei-Problem

(a) Implementieren Sie eine Queue mit den Operationen dequeue() und enqueue(key k), die intern zwei Stacks verwendet.

> **Lösung:** Idee: Unsere Queue hat intern zwei Stacke s_1 und s_2. Wir benutzen s_1, für die enqeue-Operationen und s_2 für die dequeue-Operationen. Wenn sich die Aktionen ändern, legen wir alle Elemente auf den jeweils anderen Stapel.

(b) Implementieren Sie einen Stack mit Operationen pop() und push(key k), der zwei Queues verwendet.

> **Lösung:** Idee: Unser Stack hat intern zwei Queues q_1 und q_2. Wir benutzen q_1 und q_2 jeweils abwechselnd, um die Elemente in den Stapel einzufügen. Dabei werden die Elemente bei jeder Einfügeoperation in die jeweils andere Queue kopiert, um die Reihenfolge der Elemente umzukehren. Die Lösch-Operation liest jeweils das nächste Element aus der nicht-leeren Queue.

Aufgabe 10: DivContainer-Datenstruktur

In dieser Aufgabe sollen Sie eine Datenstruktur implementieren, die über zwei Operationen verfügt: insert(x) fügt eine beliebige, positive Zahl in die Datenstruktur ein und get(d), die eine beliebige Zahl aus der Datenstruktur ausgibt, die durch d teilbar ist.

(a) Geben Sie eine Implementation beider Methoden an, wenn get(d) keine Laufzeitbeschränkungen hat und die zurückgegebene Zahl nicht aus der Datenstruktur entfernt werden soll. Welche Laufzeiten haben ihre Methoden?

> **Lösung:** Unsere Datenstruktur besitzt intern eine doppelt-verkettete Liste list, die die üblichen Methoden hat. Dann können beide Operationen durch list.insert(x) bzw. einen modifizierten list.search(k)-Algorithmus implementiert werden.

(b) Nun soll die zurückgegebene Zahl aus der Datenstruktur gelöscht werden. Wie müssen Sie Ihre Implementationen aus a) verändern, damit dies möglich wird? Ändern sich die Laufzeiten?

> **Lösung:** Wir rufen im Algorithmus die Methode list.remove(item) auf, die das Löschen für uns übernimmt. Die Laufzeit ändert sich hierdurch nicht.

Aufgabe 11: Implementieren einer eigenen Datenstruktur

Gesucht ist eine Datenstruktur MinStack zum Verwalten einer dynamischen Menge S von Zahlen. Es sollen wie bei einem Stapel die Methoden push(key k) und pop() zur Verfügung stehen, zusätzlich eine Methode Minimum(), welche die kleinste Zahl der Menge S zurück gibt. Alle Operationen sollen in konstanter Zeit ablaufen. *Tipp*: Verwenden Sie intern mehr als eine Datenstruktur.

(a) Geben Sie eine Implementierung der Datenstruktur in Pseudocode an.

> **Lösung:** Die Datenstruktur besitzt zwei Stacks s_1 und s_2. In s_1 werden ganz herkömmlich die Zahlen gespeichert. In s_2 wird eine Zahl nur dann gespeichert, falls sie das aktuelle Minimum ist. Ein Attribut minimum speichert das aktuelle Minimum.

(b) Zeigen Sie, dass es keine Datenstruktur geben kann, die zusätzlich zu den obigen Operationen eine weitere Operation popMinimum() mit konstanter Laufzeit anbietet. Diese Operation löscht das aktuelle Minimum aus dem MinStack.

Lösung: In diesem Fall ließe sich ein Sortieralgorithmus für beliebige Zahlen und mit linearer Laufzeit konstruieren, was im Widerspruch zum Resultat aus der Vorlesung steht, dass man zum Sortieren von n beliebigen Zahlen $\Omega(n \log n)$ Zeit braucht.

Aufgabe 12: Brainfuck-Interpreter (umfang- und lehrreich)

In dieser Aufgabe entwickeln wir einen Interpreter für die esoterische Programmiersprache *Brainfuck*.

(a) In unserer Version von Brainfuck besteht der Speicher aus einem Band mit theoretisch unendlich vielen Zellen, in denen ganze Zahlen stehen können. Brainfuck verwaltet einen Zeiger, der auf eine Zelle zeigt. Zu Beginn zeigt der Zeiger auf die erste Zelle des Bandes. Implementieren Sie eine Datenstruktur BFMemory, die die folgenden Operationen besitzt:

new BFMemory()	Erzeugt ein neues Band. In allen Zellen steht eine 0.
incrementPointer()	Schiebt den Zeiger auf die nächste Zelle.
decrementPointer()	Schiebt den Zeiger auf die vorherige Zelle.
incrementValue()	Erhöht den Wert der Zelle, auf der der Zeiger steht, um eins.
decrementValue()	Erniedrigt den Wert der Zelle, auf der der Zeiger steht, um eins.
int getCurrentCellValue()	Gibt den Wert der Zelle zurück, auf der der Zeiger gerade steht.

Keine der Operationen soll Fehler verursachen. Wenn der Zeiger auf eine Zelle verschoben wird, die nicht existiert, muss das Band vergrößert werden.

Lösung: Wir lösen das Problem mit einer doppelt verketteten Liste list, die als Attribut in unseren Datenstruktur-Methoden zur Verfügung steht. Am Anfgang hat die Liste einen Eintrag.

(b) Ein Brainfuck-Programm wird durch einen Array repräsentiert. Die Einträge des Arrays sind die Befehle. In unserer Brainfuck-Version erlauben wir sieben Befehle:

`>`	Verschiebt den Zeiger des Bands nach rechts.
`<`	Verschiebt den Zeiger des Bands nach links.
`+`	Inkrementiert den Wert der aktuellen Zelle.
`-`	Dekrementiert den Wert der aktuellen Zelle.
`[`	Falls der Wert der aktuellen Zelle 0 ist, springe hinter passendes `]`, ansonsten ignoriere Befehl.
`]`	Springe vor passendes `]`, welches als nächstes ausgewertet wird.
`.`	Gebe die Zelle, auf der der Zeiger steht, aus.

Welche Ausgabe hat folgendes Brainfuck-Programm?

```
++++++[>++++++++++++<-]>-.+++.<++++[>++++<-]>-.
```

Lösung: Das Programm gibt die ASCII-Werte der Buchstaben „A“, „D“ und „S“ aus, also 65, 68 und 83.

(c) Geben Sie nun unter Verwendung Ihrer Datenstruktur BFMemory einen Algorithmus an, der ein Brainfuck-Programm als Array entgegen nimmt und dieses gemäß den obigen Regeln ausführt. Sie dürfen davon ausgehen, dass die eingegebenen Programme korrekt sind. *Tipp*: Verwenden Sie Rekursion.

Lösung:
Die Schwierigkeit liegt offensichtlich in den verschachtelten Schleifen. Unser Algorithmus sucht, wenn er eine öffnende Schleife findet, nach der passenden schließenden Klammern, indem er jedes Zeichen des Codes einliest und die offenen Klammern zählt. Sobald er auf eine schließende Klammer trifft und der Zähler 0 ist, hat er die richtige Klammer gefunden. Anschließend wird diese Schleife

rekursiv ausgewertet. Um die Sache nicht unnötig zu verkomplizieren gehen wir davon aus, dass die Datenstruktur BFMemory global zugänglich ist.

(d) Sei $\mathbb{B} = \{x \in \mathbb{N} \mid 64 < x < 97\}$. Schreiben Sie einen Algorithmus, der als Eingabe eine Liste von Zahlen A mit $\mathsf{A}[i] \in \mathbb{B}$ für alle $i \leq \mathsf{A.length}$ erhält. Die Ausgabe Ihres Algorithmus soll ein gültiger Brainfuck-Code sein, der die Zahlen in A nacheinander ausgibt. Der Code muss *nicht* minimal kurz sein. Verwenden Sie eine verkettete Liste, um den Code Schritt für Schritt aufzubauen.

Lösung: Die einfachste Lösung besteht darin, lediglich eine Zelle zu verwenden und anhand der Werte im Array den Code aufzubauen.
Schneller wäre es zum Beispiel, zumindest eine konstante Schleifenkonstruktion zu Beginn einzufügen, die den Wert auf 65 hoch zählt und alle Werte in A um 65 erniedrigt. Noch besser wäre es, dynamisch Schleifen zu erzeugen, um möglichst kurzen Code zu erzeugen.