Künstliche Intelligenz
– Aufgabenblatt 01 –

Prof. Dr. David Spieler
Hochschule München

27. März 2024

Aufgabe 1 (Uninformierte Suche in Listen) *Die einfachste Suche in Listen ist die uninformierte Suche. Die Liste wird hier von Anfang bis Ende Element für Element durchgegangen, wobei das aktuelle Element jeweils mit dem gesuchten Wert verglichen wird.*

1. *Implementieren Sie eine Funktion* `search(l, v)`*, die in Liste* `l` *auf diese Weise den Wert* `v` *sucht. Wird das Element gefunden, wird es zurückgegeben. Andernfalls wird* `None` *zurückgegeben.*

2. *Angenommen die Liste hat n Elemente, wieviele Elemente müssen hier minimal, maximal und durchschnittlich untersucht werden?*

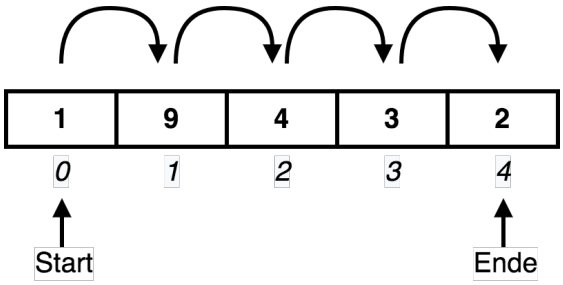

Abbildung 1: Uninformierte Suche

Aufgabe 2 (Binäre Suche) *Weiß man, dass die zu durchsuchende Liste sortiert ist, also dass die Elemente der Liste in aufsteigender Reihenfolge angeordnet sind, kann man die Suche nach einem Wert mit Hilfe der Binärsuche effizienter gestalten. Das bedeutet, es müssen nicht mehr zwangsläufig alle Elemente durchgegangen werden. Der Trick dabei ist, in der Mitte der Liste mit der Suche zu beginnen. Befindet sich hier bereits der*

gesuchte Wert, ist die Suche zu Ende. Ist der Wert in der Mitte kleiner als der gesuchte Wert, so muss in der rechten Teilliste weitergesucht werden, andernfalls in der linken Teilliste. Dieser Vorgang wird nun rekursiv wiederholt, indem die gleiche Strategie in der entsprechenden (maximal halb so großen) Teilliste angewandt wird – und wieder und immer wieder – bis entweder der Wert gefunden wurde oder die zu durchsuchende Liste zu klein wird.

1. *Implementieren Sie eine Funktion* `binsearch(l, v)`*, die in Liste* `l` *auf diese Weise den Wert* `v` *sucht. Wird das Element gefunden, wird es zurückgegeben. Andernfalls wird* `None` *zurückgegeben.*

2. *Angenommen die Liste hat n Elemente, wieviele Elemente müssen hier minimal, maximal und durchschnittlich untersucht werden?*

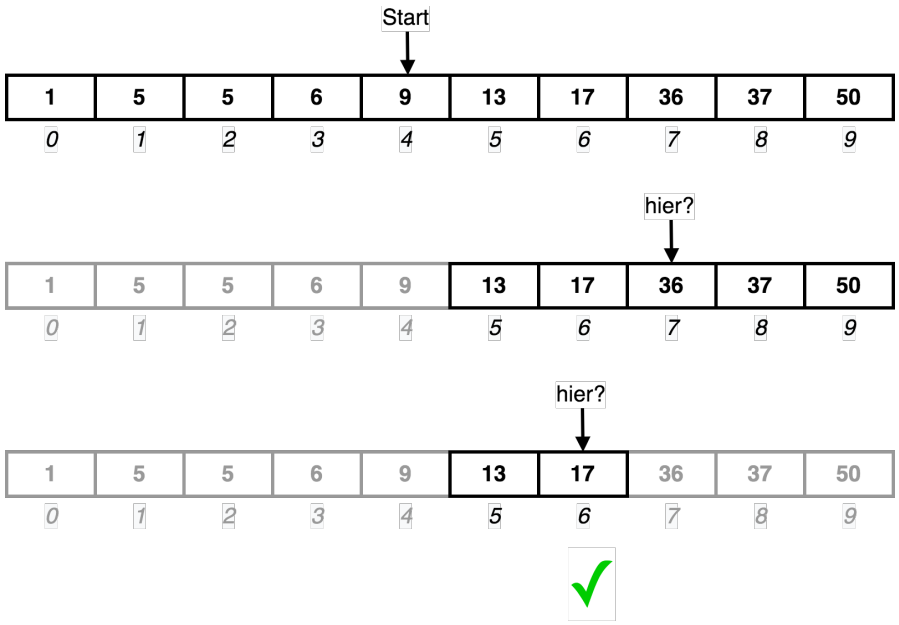

Abbildung 2: Binäre Suche

Aufgabe 3 (Umkehrung von Listen mit Stacks) *Implementieren Sie eine Funktion* `reverse(l)`*, welche eine Liste* `l` *als Argument annimmt und in umgedrehter Reihenfolge wieder zurückgibt. Verwenden Sie dafür einen Stack, also wiederum eine temporäre Liste, bei der Sie die Methoden* `append` *und* `pop` *nutzen. Hinweis: Sie können mit* `len(s) > 0` *überprüfen, ob ein/-e Stack/Liste* `s` *noch Elemente beinhaltet.*

Aufgabe 4 (Kryptographie mit Maps) *Maps funktionieren wie Wörterbücher – ein Schlüssel wird übersetzt zum entsprechenden Wert. Dieses Prinzip lässt sich auch auf*

einzelne Buchstaben anwenden und man erhält damit eine einfache Verschlüsselung mit Hilfe der Substitutionsmethode. Wir wollen ein solches System nun in Python programmieren.

1. *Erstellen Sie eine Map* `cipher`*, welche die Verschlüsselungstabelle aus Abbildung 3 kodiert.*

2. *Implementieren Sie eine Funktion* `transform(data, table)`*, welche eine Zeichenkette* `data` *mit Hilfe einer Verschlüsselungstabelle* `table` *Buchstabe für Buchstabe übersetzt und das Ergebnis zurückgibt.*

3. *Verschlüsseln Sie die Zeichenkette* `'hallo welt, ich bin streng geheim.'` *(Achtung: Alle Buchstaben sind klein geschrieben!) mit Hilfe der Methode* `transform` *und der Verschlüsselungstabelle* `cipher`*. Speicher Sie das Ergebnis in der Variablen* `secret`*. Können Sie den Satz in* `secret` *lesen?*

4. *Implementieren Sie eine Funktion* `invert(table)`*, welche eine Verschlüsselungstabelle* `table` *nimmt und deren Inverse berechnet. Gibt es beispielsweise eine Zuordnung* `'a': 'b'` *in der ursprünglichen Tabelle, soll das Ergebnis eine Zuordnung* `'b': 'a'` *beinhalten. Hinweis: Die Menge aller Schlüssel in einer Map erhalten Sie mit der Methode* `keys()`*.*

5. *Berechnen Sie die Inverse von* `cipher` *und legen Sie das Ergebnis in* `inverseCipher` *ab.*

6. *Führen Sie* `transform(secret, inverseCipher)` *aus. Was ist passiert?*

Schlüssel	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z		.	,
Wert	c	l	b	t	g	w	f	r		k	q	a	,	j	.	y	e	x	o	z	i	n	v	s	d	m	h	u	p

Abbildung 3: Verschlüsselungstabelle

Aufgabe 5 (Bäume) *In dieser Aufgabe wollen wir mit der Datenstruktur Baum experimentieren.*

1. *Implementieren Sie die zunächst die Baumdatenstruktur mit Hilfe der Klasse* `Tree` *aus dem Foliensatz.*

2. *Erstellen Sie den Beispielbaum aus dem Foliensatz mit Hilfe der Klasse* `Tree`*.*

3. *Was macht die Methode* `printTree`*? Probieren Sie Methode an dem Baum aus dem Foliensatz aus.*

```
def printTree(tree, level=0):
    print("  " * level, tree.data)
    for child in tree.children:
        printTree(child, level + 1)
```

Aufgabe 6 (Graphen) *In dieser Aufgabe wollen wir mit der Datenstruktur Graph experimentieren.*

1. *Implementieren Sie die zunächst die Graphdatenstruktur mit Hilfe der Klasse* `Graph` *aus dem Foliensatz.*

2. *Erweitern Sie die Klasse* `Graph` *um eine Methode* `addVertex(v)`*, die einen neuen Knoten* `v` *hinzufügt.*

3. *Erweitern Sie die Klasse* `Graph` *um eine Methode* `addEdge(f, t)`*, die eine neue Kante* `(f, t)` *hinzufügt.*

4. *Erweitern Sie die Klasse* `Graph` *um die beiden Methoden* `getSuccessors(v)` *und* `getPredecessors(v)`*, welche die Menge der Nachfolger bzw. Vorgänger von Knoten* `v` *berechnen.*

5. *Testen Sie die neuen Methoden am Beispielgraphen aus dem Foliensatz. Hinweis: Sie können sich die Knoten- und Kantenmenge eines Graphen* `g` *mit* `g.vertices` *bzw.* `g.edges` *ansehen.*