Informatik 5. Klasse

Examensvorbereitung

Teil 3

Lösungen

Suchen: Aufgabe 1

```python
def linearsearch(L, item):
    pos = []
    for i in range(0, len(L)):
        if L[i] == item:
            pos.append(i)
    return pos

L = [1, 0, 1, 1, 0]
print(linearsearch(L, 1)) # => [0, 2, 3]
print(linearsearch(L, 2)) # => []

# Oder für Furchtlose als "Einzeiler" mit einer List Comprehension:

def linearsearch2(L, item):
    return [i for i in range(0, len(L)) if L[i] == item]

# Hinweis: statt range(0, len(L)) kann auch kürzer range(len(L))
# geschrieben werden. Python geht dann automatisch davon aus,
# dass der erste Index null ist.
```

Suchen: Aufgabe 2

Suche das Element 28 in der folgenden Liste:

0	1	2	3	4	5	6	7
3	5	7	11	15	21	28	33

$$i = 0,\ j = 7 \Rightarrow m = \left\lfloor \frac{i+j}{2} \right\rfloor = 3;\ A[3] = 11 < 28 \Rightarrow i = m + 1 = 4$$

$$i = 4,\ j = 7 \Rightarrow m = \left\lfloor \frac{i+j}{2} \right\rfloor = 5;\ A[5] = 21 < 28 \Rightarrow i = m + 1 = 6$$

$$i = 6,\ j = 7 \Rightarrow m = \left\lfloor \frac{i+j}{2} \right\rfloor = 6;\ A[6] = 28 = 28 \Rightarrow 28 \in A \text{ (Ende)}$$

Suchen: Aufgabe 3

Suche das Element 6 in der folgenden Liste:

0	1	2	3	4	5	6	7
3	5	7	11	15	21	28	33

$i = 0,\ j = 7 \Rightarrow m = \left\lfloor \frac{i+j}{2} \right\rfloor = 3;\ A[3] = 11 > 6 \Rightarrow j = m - 1 = 2$

$i = 0,\ j = 2 \Rightarrow m = \left\lfloor \frac{i+j}{2} \right\rfloor = 1;\ A[1] = 5 < 6 \Rightarrow i = m + 1 = 2$

$i = 2,\ j = 2 \Rightarrow m = \left\lfloor \frac{i+j}{2} \right\rfloor = 2;\ A[2] = 7 > 6 \Rightarrow j = m - 1 = 1$

die untere Grenze $i = 2$ ist grösser als die obere Grenze $j = 1 \Rightarrow 6 \notin A$

Suchen: Aufgabe 4

Für die binäre Suche muss eine Liste aufsteigend sortiert sein.

Suchen: Aufgabe 5

Laufzeitkomplexität der linearen Suche nach einem Element x in einer Liste L.

- *Best Case*: Wenn sich das gesuchte Element am Anfang oder immer unter den ersten k Elementen befindet, wobei k eine feste Konstante ist. Dann muss immer ein konstanter Aufwand $T(n) \leq C \cdot k$ betrieben werden, um das Element x zu finden $\Rightarrow O(1)$.
- *Average Case*: Wenn sich das gesuchte Element mit gleicher Wahrscheinlichkeit an jeder Postition befindet, dann betragen die Kosten für das Auffinden von x im Mittel $T(n) = C \cdot \frac{1}{2}n \Rightarrow O(1)$.
- *Worst Case*: Wenn sich das gesuchte Element nicht in der Liste oder am Ende der Liste befindet (oder unter den k letzten, wobei k eine Konstante ist), dann betragen die Laufzeitrelevanten Kosten für das (nicht) Auffinden $T(n) = C \cdot n \Rightarrow O(n)$.

Suchen: Aufgabe 6

Worst Case: Das gesuchte Element befindet sich nicht in der Liste oder wird erst im letzten möglichen Suchschritt (k) gefunden.

Suchschritt	Länge der verbleibenden Teilliste
1	$n/2 = n/2^1$
2	$n/4 = n/2^2$
3	$n/8 = n/2^3$
...	...
k	$n/2^k$

Im Worst Case hat die zu durchsuchende Liste noch die Länge 1. Dann wissen wir, ob sie das gesuchte Element enthält oder nicht und können die Suche beenden.

$$1 = \frac{n}{2^k} \quad \text{(siehe obige Bemerkung)}$$

$$2^k = n \quad \text{(Algebra ...)}$$

$$\log_2(2^k) = \log_2(n)$$

$$k = \log_2(n)$$

Best Case: Das gesuchte Element befindet sich in der „Mitte“ der Liste; denn dort startet die binäre Suche, indem sie das Element an der Position `(L[0] + L[n-1])//2` testet.

Suchen: Aufgabe 7

Es lohnt sich nicht, denn die schnellsten (vergleichsbasierten) Sortierverfahren haben die Laufzeitkomplexität $O(n \log(n))$, während die naive lineare Suche eine Laufzeitkomplexität von $O(n)$ hat. Wegen $n < n \log(n)$ (für $n > 1$) wäre die lineare Suche asymptotisch (d. h. für grosse n) bereits fertig, bevor die Liste sortiert wurde.

Suchen: Aufgabe 8

(a) Beim *String-Matching* möchte man wissen, ob oder wo eine Zeichenfolge p (*pattern*) in einer Zeichefolge t (*text*) enthalten ist, wobei wir sinnvollerweise voraussetzen, dass p kürzer als t ist.

(b)
- Suche nach Textstellen in einer Textverarbeitung oder im Internet
- Analyse von DNA- oder Proteinsequenzen (Biologie)
- Erkennen von Plagiaten (Fremde geistige Leistungen als eigene ausgegeben)
- Virenscanner

Suchen: Aufgabe 9

1	0	1	0	0	1	0	1	1	0	
1	0	1	1							4
	1	0	1	1						1
		1	0	1	1					3
			1	0	1	1				1
				1	0	1	1			1
					1	0	1	1		4

Insgesamt 14 Vergleiche

Grundsätzlich kosten auch die "Verschiebungen" des Musters etwas Zeit. Da nach jeder Vergrösserung der Startposition des Musters um 1 mindestens ein Vergleich gemacht wird, ist die Anzahl der Verschiebungen immer kleiner oder gleich der Anzahl der Vergleiche und hat deshalb keinen Einfluss auf die asymptotische Laufzeitkomplexität der naiven Suche – nur auf die Proportionalitätskonstante in der Kostenfunktion $T(n)$.

Suchen: Aufgabe 10

Gegeben: Muster p mit $|p| = m$
Text t mit $|t| = n$

- *Best Case:* $O(m)$

  Beispiel: `p = aab`, `t = aabaaaaaaa`

  Wenn das Muster gleich zu Beginn des Textes steht und der Algorithmus für jedes Zeichen des Musters p einen Vergleich durchführen muss, ist er nach $1 \times m = m$ Vergleichen fertig.

- *Worst Case:* $O(nm)$

  Beispiel: `p = aab`, `t = aaaaaaaaab`

  Die naive Auche muss an jeder neuen Position m Vergleiche durchführen, da die Ungleichheit erst an der letzten Position festgetellt wird.

  Da es $(n-m+1)$ Positionen, gibt, an denen diese m Vergleiche durchgeführt, werden beträgt die Anzahl der Vergleiche insgesamt:

  $(n - m + 1) \times m = nm - m^2 + m \approx nm$, wenn $m \ll n$

Suchen: Aufgabe 11

Bad Match Table (BMT) für das Muster `p=GCTT`:

1. Eine Kolonne pro *verschiedenes* Zeichen in p und einmal `*` für alle andern Zeichen im Alphabet.
2. In der 2. Zeile zunächst überall die Länge des Musters eintragen.
3. p vom ersten bis zum *zweitletzten* Zeichen z durchlaufen und die unterste Zahl in der Kolonne von z durch die Distanz zum letzten Zeichen ersetzen.
4. Die jeweils letzten Zahlen geben die Verschiebung für das entsprechende Zeichen an (siehe unten).

G	C	T	*
~~4~~	~~4~~	~~4~~	4
3	2	1	

1. Schreibe das Muster linksbündig unter den Text;
2. Vergleiche Muster mit Text *von rechts nach links*. (Vergleiche sind hier rot gekennzeichnet).
3. Stoppe, wenn alle Zeichen übereinstimmen → gefunden!
4. Sonst markiere das Zeichen z *im Text*, das über dem letzten Musterzeichen steht (hier: eingekreist).
5. Verschiebe das Muster gemäss z in BMT und fahre bei Schritt 2 fort.

C	C	G	(T)	(G)	G	T	(A)	G	C	T	(T)	Vergleiche
G	C	T	T									2
	G	C	T	T								1
				G	C	T	T					1
								G	C	T	T	4
												8

Suchen: Aufgabe 12

(a) Die Antwort auf die Frage lautet: Es kommt darauf an.

Die Schnelligkeit ist von einem Zusammenspiel der folgenden Faktoren abhängig:

- dem Alphabet
- der Länge von Muster und Text
- der Struktur von Muster und Text
- dem Algorithmus

(b) Vergleichen wird die Algorithmen miteinander, erkennen wir dass für kurze Muster p mit $|p| = 2, 3$ das naive Verfahren schneller ist als BMH. Für längere Muster ($|p| = 4, 5, 6, 7$) ist hingegen BMH schneller.

Interpretation: BMH muss vor der Suche noch eine Tabelle aufbauen, mit der grössere Verschiebungen als nur ein Zeichen möglich sind. Dieser Aufbau kostet Zeit und kann bei kurzen Zeichenfolgen nicht amortisert werden. Umgekehrt ist der naive Algorithmus kürzer und kann daher bei kurzen Mustern diesen Vorteil besser ausspielen.

TSP: Aufgabe 1

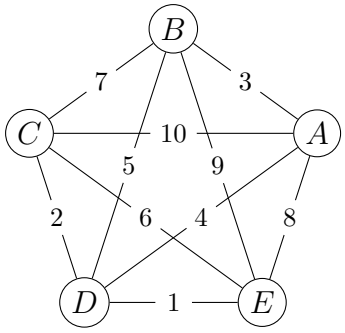
- Logistik: Verteilen von Paketen durch einen Paketboten
- Steuerung eines Bohrers für die Löcher einer Leiterplatte
- Biologie: Auffinden einer DNA-Sequenz minimaler Länge, die eine Menge maximal überlappender Teilsequenzen enthält

TSP: Aufgabe 2

Wenn wir ein einer der 5 Städte beginnen, dann können wir die verbleibenden 4 Städte auf $4! = 24$ Arten durchlaufen.

Zählt man die jeweilige Reise in umgekehrter Richung nicht mit, dann sind es nur noch $24/2 = 12$ Rundreisen.

TSP: Aufgabe 3



	A	B	C	D	E
A	0	3	10	4	8
B	3	0	7	5	9
C	10	7	0	2	6
D	4	5	2	0	1
E	8	9	6	1	0

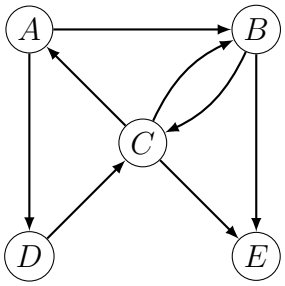
TSP: Aufgabe 4

Hier sind gleich drei Lösungsvarianten zur Auswahl:

```
# iterativ
def factorial_a(n):
    f = 1
    for k in range(1, n+1):
        f *= k
    return f

# rekursiv (funktioniert bis n=997 => maximale Rekurrsionstiefe)
def factorial_b(n):
    if n == 0:
        return 1
    else:
        return n*factorial_b(n-1)

# lazy!
from math import factorial as factorial_c

# Testcode (nicht verlangt)
for n in range(0, 10):
    print(n, factorial_a(n), factorial_b(n), factorial_c(n))

# Output der Tests:
# 0 1 1 1
# 1 1 1 1
# 2 2 2 2
# 3 6 6 6
# 4 24 24 24
# 5 120 120 120
# 6 720 720 720
# 7 5040 5040 5040
# 8 40320 40320 40320
# 9 362880 362880 362880
```

TSP: Aufgabe 5

(a) Graph: (die bildliche Darstellung ist nicht eindeutig)



(b) Zyklen sind $ADCA$ und $ABCA$

TSP: Aufgabe 6

$$\frac{n(n-1)}{2} = 78$$
$$n(n-1) = 156$$
$$\ldots = \ldots$$

Eigentlich müsste man die quadratische Gleichung lösen, aber da n und $n-1$ so nahe beieinander liegen, kann man auch die Wurzel aus 156 ziehen und das Resultat auf- sowie abrunden, um die beiden benachbarten Zahlen zu erhalten. Es geht noch einfacher im Kopf, indem man die beiden Quadratzahlen errät die kleiner und grösser als 156 sind: $12^2 = 144 < 156 < 169 = 13^2$ und die Wurzel der grösseren nimmt. Somit muss der Graph $n = 13$ Knoten haben.

TSP: Aufgabe 7

(a) Ohne Beschränkung der Allgemeinheit, können wir willkürlich eine der n Städte als Startknoten wählen.

(b) Von dieser Stadt aus, gibt es $(n-1)!$ mögliche Rundtouren.

(c) Zur Berechnung jeder dieser Rundtouren müssen wir aus einer Distanzatrix (Distanztabelle) n Distanzen auslesen und addieren. Dies ist mit einem Aufwand von $O(n)$ verbunden

(d) Aus (b) und (c) folgt, dass die Laufzeitkomplexität für alle Touren insgesamt
$(n-1)! \cdot O(n) = O(n(n-1)!) = O(n!)$
betragen.

Für ein symmetrisches TSP können wir die Kosten noch halbieren, was zwar einen Einfluss auf die konkrete Dauer der Lösung hat aber nicht auf die Laufzeitkomplexität, da konstante Faktoren weggelasssen werden: $O(\frac{1}{2}n!) = O(n!)$.

TSP: Aufgabe 8

	A	B	C	D
A	0	1	2	2
B	1	0	6	3
C	2	6	0	11
D	2	3	11	0

$ABCDA \quad \Rightarrow \quad 20$

$ABDCA \quad \Rightarrow \quad 17$

$ACBDA \quad \Rightarrow \quad 13$ optimale Route

$ACDBA \sim ABDCA$

$ADBCA \sim ACBDA$

$ADCBA \sim ABCDA$

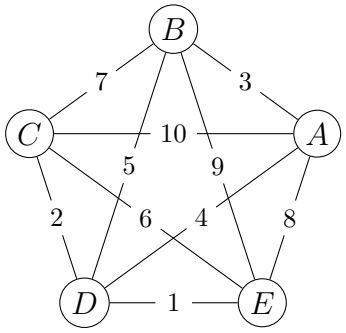
TSP: Aufgabe 9

$T(12) = C \cdot 12! = 30\,\text{s}$

$T(14) = C \cdot 14! = 14 \cdot 13 \cdot C \cdot 12! = 182 \cdot 30\,\text{s} = 5460\,\text{s} = 91\,\text{Min}$

TSP: Aufgabe 10

(a) Graph



NNH: $A \xrightarrow{3} B \xrightarrow{5} D \xrightarrow{1} E \xrightarrow{6} C \xrightarrow{10} A \quad \Rightarrow \quad$ Länge: 25

kürzere Rundtouren:

$A-B-C-D-E-A$: $3+7+2+1+8=21$

$A-B-C-E-D-A$: $3+7+6+1+4=21$

$A-B-D-C-E-A$: $3+5+2+6+8=24$

$A-B-E-C-D-A$: $3+9+6+2+4=24$

(b)
- Vorteil NNH: ist effizenter als Brute Force (polynomielle Laufzeit)
- Nachteil NNH: liefert im Allgemeinen keine optimale Route