Aufgabe 1.2

Entscheiden Sie die Gültigkeit der folgenden Aussagen (nicht notwendigerweise formal; sie dürfen auch intuitiv argumentieren):

(a) $n^{100} = O(1.01^n)$

(b) $10^{\log n} = O(2^n)$

(c) $10^{\sqrt{n}} = O(2^n)$

(d) $10^n = O(2^n)$

Lösung

(a) Stimmt – wenn man n_0 groß genug wählt. Exponentialfunktionen (mit Basis > 1 wachsen immer schneller als Polynome.

(b) Stimmt – $\log n$ wächst wesentlich langsamer als n und damit wächst auch $10^{\log n}$ wesentlich langsamer

(c) Stimmt – denn $10^{\sqrt{n}} = (2^{\log_2 10})^{2^{\sqrt{n}}} = 2^{\log_2 10 \cdot \sqrt{n}}$ und das liegt in $O(2^{\sqrt{n} \cdot \sqrt{n}}$, denn $\log_2 19$ ist ja nur eine Konstante und wird ab einem bestimmten n_0 kleiner sein als $\sqrt{n_0}$.

(d) Stimmt nicht – $10^n = 2^{\log_2 10 \ \cdot n}$ wird immer schneller wachsen als 2^n. Die Konstante $\log_2 10$ im Exponenten lässt sich nicht durch eine große Konstante C vor dem Term 2^n ausbügeln.

Aufgabe 1.5

Verwenden Sie die Python-Funktion *reduce*, um eine Funktion *prod*(*lst*) zu definieren, die als Ergebnis die Aufmultiplikation der Zahlen in *lst* zurückliefert. Mathematisch ausgedrückt, sollte für *prod* gelten:

$$\texttt{prod(lst)} \stackrel{!}{=} \prod_{x \in \texttt{lst}} x$$

Implementieren Sie nun *facIter* mit Hilfe von *prod*.

Lösung

Ein iterative Implementierung der Fakultätsfunktion mittels Listenkomprehensionen:

```
def prod( lst ):
  return reduce(lambda x,y:x*y, lst, 1)

def facIter(n):
  return prod(range(1,n+1))
```

Aufgabe 1.6

Angenommen, eine rekursive Funktion erhält als Argument eine reelle Zahl. Warum ist es für eine korrekt funktionierende rekursive Funktion nicht ausreichend zu fordern, dass die rekursiven Aufrufe als Argumente kleinere reelle Zahlen erhalten als die aufrufende Funktion?

Lösung

Der Grund ist: es gibt zu viele Fließkommazahlen; wieviele es tatsächlich gibt hängt davon ab, wieviele Bits für die Darstellung der Fließkommazahl verwendet werden. Nehmen wir an, es werden 64 Bits verwendet... 52 Mantissenbits, 11 Exponentenbits und ein Vorzeichenbit. Es gibt daher allein schon 2^{52} Fließkommazahlen zwischen 0 und 1; das sind zwar „nur“ endlich viele, praktisch würde der rekursive Abstieg aber einer Endlosschleife gleichen.

Aufgabe 1.7

(a) Definieren Sie die Funktion $sum(n)$, die die Summe der Zahlen von 1 bis n berechnen soll, rekursiv.

(b) Definieren Sie die Funktion $len(\,lst\,)$, die die Länge der Liste lst berechnen soll, rekursiv.

Lösung

(a) Definition der Funktion *sum* rekursiv:

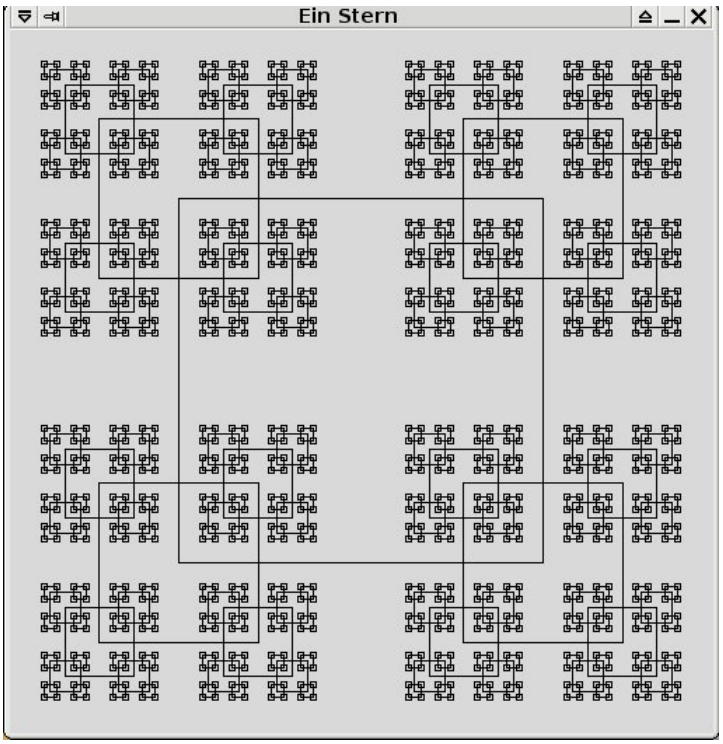
```
1 def mysum(n):
2   if n==0:
3     return 0
4   else:
5     return n + mysum(n -1)
```

(b) Definition der Funktion *len* rekursiv:

```
1 def mylen(lst):
2   if lst==[]:
3     return 0
4   else:
5     return 1 + mylen(lst[1:])
```

Aufgabe 1.9

Zeichnen Sie durch eine rekursiv definierte Python-Funktion und unter Verwendung der *graphics*-Bibliothek folgenden Stern:



Lösung

Durch die Python-Funktion *star* in folgendem Listing kann der rekursive Stern gezeichnet werden:

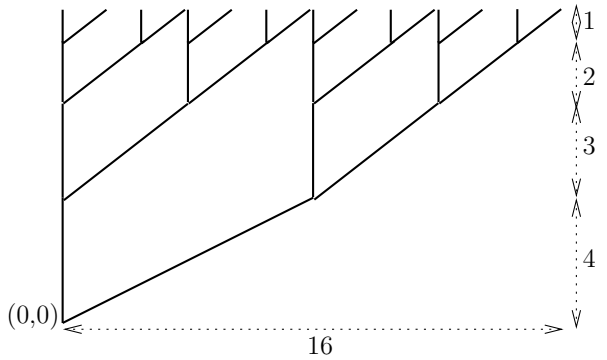
```
1  from graphics import *
2
3  starCanv = GraphWin("Ein Stern",500,500)
4
5  def box(x,y,r):
6      p1 = Point(x-r,y-r)
7      p2 = Point(x+r,y+r)
8      r = Rectangle(p1,p2)
9      r.draw(starCanv)
10
11 def star(x,y,r):
12     f=2.3
13     if r<=2: return
14     box(x,y,r)
```

```
15    star(x-r,y-r,r/f)
16    star(x+r,y-r,r/f)
17    star(x+r,y+r,r/f)
18    star(x-r,y+r,r/f)
```

Die Funktion *box* zeichnet ein Quadrat mit Zentrum (x, y) und „Radius“ r. Mit dem Faktor f=2.3 kann man einstellen, wie „luftig“ der Stern aussehen soll – je größer f, desto mehr Platz ist zwischen den einzelnen Quadraten.

Aufgabe 1.10

Schreiben Sie eine rekursive Prozedur *baum*(*x*,*y*,*b*,*h*) zum Zeichnen eines (binären) Baumes derart, dass die Wurzel sich bei (x, y) befindet, der Baum b breit und h hoch ist. Definieren Sie hierzu eine Python-Prozedur *line*(*x1*,*y2*,*x2*,*y2*), die eine Linie vom Punkt (*x1*,*y2*) zum Punkt (*x2*,*y2*) zeichnet. Folgende Abbildung zeigt ein Beispiel für die Ausgabe die der Aufruf *baum*(0,0,16,4) erzeugt.



Lösung

```
1  def baum(x, y, b, h):
2    m = x + (y-x) / 2
3    if h>0:
4      line(x,y,x,y+h)
5      line(x,y,x+m,y+h)
6      baum(x,y+h,b/2,h-1)
7      baum(x+m,y+h,b/2,h-1)
```

Das Schreiben der Prozedur *line* überlassen wir dem Leser.

Aufgabe 2.1

Implementieren Sie – ebenfalls unter Verwendung von *insND* – eine iterative Variante von *insertionSortRek*.

Lösung

Eine iterative Variante basierend auf der in Listing **??** gezeigten Funktion *insAtRightPosND*. Verwendet wird eine **for**-Schleife die über alle Listenelemente läuft:

```
1 def insertionSort2( alst ):
2   for i in range(1,len( alst )):
3       alst [0:i+1]=insAtRightPosND(alst[0:i],alst[i])
```

Listing 1: Implementierung von Insertion Sort – iterativ

Aufgabe 2.2

Die Funktion *insertionSort* durchsucht die bereits sortierte Liste *linear* nach der Position, an die das nächste Element eingefügt werden kann. Kann man die Laufzeit von *insertionSort* dadurch verbessern, dass man eine binäre Suche zur Bestimmung der Einfügeposition verwendet, die Suche also in der Mitte der sortierten Teilliste beginnen lässt und dann, abhängig davon, ob der Wert dort größer oder kleiner als der einzufügende Wert ist, in der linken bzw. rechten Hälfte weitersucht, usw.?

Falls ja: Was hätte solch ein Insertion-Sort-Algorithmus für eine Laufzeit? Implementieren Sie Insertion Sort mit binärer Suche.

Lösung

Ein binäre Suche in einem sortierten Array benötigt tatsächlich nur eine logarithmische Suchzeit. Die Zeit, ein Element in ein solches Array einzufügen kostet jedoch $O(n)$ Zeit, da alle Elemente nach der Einfügeposition eine Position weiter geschoben werden müssen; im Mittel sind dies $n/2$ Elemente, daher die Laufzeitkomplexität von $O(n)$.

Es sei hier jedoch folgendes angemerkt: Es ist eine Implementierung eines Arrays (bzw. einer Liste) denkbar, mit der man eine deutlich bessere amortisierte Zeit erreichen kann: Sei die binäre Darstellung der Zahl n gegeben durch

$$n_{k-1}n_{k-2}\ldots n_1n_0$$

Man verwende k sortierte Arrays $A_0,\ldots A_{k-1}$; ist das Bit n_l gesetzt, so enthält das Array A_l genau 2^l Elemente, andernfalls ist es leer. Die Gesamtzahl der in den Arrays befindlichen Elementen ist somit $\sum_{l=0}^{k-1} n_l 2^l = n$. Man kann nun eine Such-, Einfüge und Löschopertion implementieren, die jeweils eine amortisierte Laufzeitkomplexität von

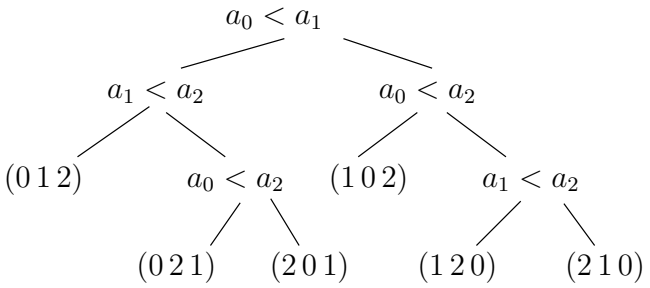
$O(\log n)$ haben. Es sei an dieser Stelle dem Leser überlassen, sich über die Details einer solchen Implementierung Gedanken zu machen.

Aufgabe 2.3

Erstellen Sie einen Entscheidungsbaum, der die Funktionsweise von Insertion Sort beschreibt, zum Sortieren einer 3-elementigen Liste.

Lösung

Folgender Entscheidungsbaum modelliert das Verhalten von Insertion Sort bei Eingabe einer 3-Elementigen Liste $[a_0, a_1, a_2]$



Aufgabe 2.4

Würde Insertion Sort, was die getätigten Vergleiche betrifft, so vorgehen, wie durch den in Abbildung **??** gezeigten Entscheidungsbaum beschrieben?

Lösung

Der Vergleich an der Wurzel wäre in der Tat der erste Vergleich, den ein Insertion Sort Algorithmus durchführen würde. Denn: a_0 ist im ersten Schleifendurchlauf das einzige Element des schon sortieren Teils der Liste; das erste Element des unsortierten Teils der Liste, also a_1 wird anfänglich mit a_0 verglichen.

Aber schon der „erste“ Vergleich im rechten Teilbaum passt nicht: Im rechten Teilbaum gilt, dass $a_0 \geq a_1$, h. h. der schon sortierte Teil der Liste wäre nach diesem Schritt $[a_1, a_0$. Die in Listing **??** gezeigte Implementierung *insertionSort* geht den sortierten Teil der Liste von rechts nach links, beginnend mit dem größten Element, durch und sucht nach der richtigen Einfügeposition. D.h. a_2 sollte also an dieser Stelle mit dem größten Element der schon sortierten Teilliste, nämlich a_0 verglichen werden. Die Wurzel des rechten Teilbaums sollte also mit „$a_0 < a_2$“ markiert sein.

Aufgabe 2.5

Um *quicksort* noch kompakter zu implementieren, verwenden Sie die Hilfsfunktion:

def $o(x,s)$: **return** $[i$ **for** i **in** x **if** $cmp(i,x[0])==s]$

(ein Funktions-Body mit nur einer Zeile ist möglich!)

Lösung

Mit Hilfe der Funktion

def $o(x,s)$: **return** $[i$ **for** i **in** x **if** $cmp(i,x[0])==s]$

kann der Quicksort-Algorithmus in nur einer Zeile implementiert werden:

```
def qs(x):
  return len(x)>1 and qs(o(x,-1)) + o(x,0) + qs(x,o(1)) or x
```

Aufgabe 2.6

Implementieren Sie eine randomisierte Variante von Quicksort

quicksortRandomisiert(*lst* , *l* , *r*)

die eine Häufung ungünstiger Fälle dadurch vermeidet, dass das Pivot-Element der Partitionierung von *lst* [*l* :*r*+1] zufällig aus den Indizes zwischen (einschließlich) *l* und *r* gewählt wird.

Lösung

Verändert wird bei der randomisierten Variante von Quicksort eigentlich nur die Art und Weise wie eine Liste partitioniert wird. Dies geschieht jetzt folgendermaßen:

```
1  def parIPRandom(lst,l,r):
2    randInd = random.randint(l,r)
3    lst [ l ], lst [randInd]=lst[randInd], lst [ l ]
4    x=lst[ l ]
5    i=l-1
6    j=r+1
7    while True:
8      while True:
9        j=j-1
10       if lst [j]≤x: break
11     while True:
12       i=i+1
13       if lst [i]≥x: break
14     if i<j:
15       lst [i], lst [j]=lst[j], lst [i]
```

```
16      else:
17        return j
```

In Zeile 2 wird ein zufälliger Index *randInd* zwischen einschließlich l und r ausgewählt. In Zeile 3 wird das linke Element der Liste *lst* mit dem am Index *randInd* befindlichen Index getauscht und dann einfach wie im nicht-randomisierten Fall fortgefahren, wobei das Pivot-Element nun zufällig gewählt ist.

Der eigentliche Quicksort-Algorithmus nimmt dann einfach statt der *partitionIP*-Funktion die randomisierte Variante *parIPRandom*.

Aufgabe 2.7

Implementieren Sie eine weitere randomisierte Variante von Quicksort

$$quicksortMedian(lst, l, r)$$

die das Pivotelement folgendermaßen wählt:

- Es werden zunächst drei zufällige Elemente aus der zu partitionierenden Liste (also aus $lst[l:r+1]$) gewählt.
- Als Pivot-Element wird der Median – also das mittlere der zufällig gewählten Elemente – ausgewählt.

Lösung

Die Wahl des Median erfordert etwas mehr Code als im einfachen randomisierten Fall:

```
1  def parIPMedian(lst,l,r):
2    if r-l<=2: # Es gibt keine drei Elemente
3      randInd = random.randint(l,r)
4    else:
5      inds = random.sample(xrange(l,r+1),3) # 3 auswaehlen
6      vals=[(lst[i],i) for i in inds]        # dekoriert mit den Werten
7      vals.sort()
8      randInd=vals[1][1]                     # den mittleren auswaehlen
9    lst[l], lst[randInd]=lst[randInd],lst[l]
10   ...
```

Zunächst muss geprüft werden, ob der zu partitionierende Bereich überhaupt 3 oder mehr Elemente besitzt. Falls nicht, so kann auch kein Median bestimmt werden und es wird einfach aus den 1 oder 2 Elementen, die die Liste in diesem Falle hat, ein zufälliger Index ausgewählt und in *randInt* gespeichert (Zeile 3).

Falls die Liste drei oder mehr Elemente besitzt, so werden mittels *random.sample* zufällig drei verschiedene Indizes aus dem Bereich zwischen einschließlich l und r ausgewählt (Zeile 5). Die an den drei Positionen befindlichen Werte werden sortiert und

schließlich wird mittels *vals*[1][1] der Index des mittleren Elements ausgewählt und in *randInd* gespeichert.

Aufgabe 2.8

Vergleichen Sie nun die Algorithmen *quicksortIP*, *quicksortRandomisiert* und *quicksortMedian* folgendermaßen:

- Generieren Sie 100 zufällig erzeugte 10.000-elementige Listen, die Werte aus $\{1, \ldots 100.000\}$ enthalten und lassen sie diese 100 Listen durch die drei Quicksort-Varianten sortieren.
- „Merken“ Sie sich für jeden der Algorithmen jeweils die folgenden Daten:
  1. Die durchschnittliche Zeit, die der jeweilige Algorithmus zum Sortieren einer 10.000-elementigen Liste brauchte.
  2. Die – aus den 100 Sortierdurchläufen – schnellste Zeit, die der jeweilige Algorithmus zum Sortieren einer 10.000-elementigen Liste brauchte.
  3. Die – aus den 100 Sortierdurchläufen – langsamste Zeit, die der jeweilige Algorithmus zum Sortieren einer 10.000-elementigen Liste brauchte.

Bemerkung: Zum Erzeugen einer Liste mit zufällig gewählten Elementen können Sie das Python-Modul *random* verwenden. Der Aufruf *random.randint*(*a*,*b*) liefert eine zufällige *int*-Zahl zwischen einschließlich *a* und *b* zurück.
Zur Zeitmessung können Sie das Python-Modul *time* verwenden. Der Aufruf *time.time*() (unter Windows besser: *time.clock*()) liefert die aktuelle CPU-Zeit zurück.

Lösung

Das Benchmarking des Quicksort und der beiden randomisierten Varianten kann elegant über die folgende Funktion *benchmarkAlg* gelöst werden, die als Argument ein Sortier-Funktion erwartet.

```
1  def benchmarkAlg(alg):
2    groesse=10000
3    n=100
4    zeiten = []
5    for i in range(1,n+1):
6      lst=randlst(groesse)
7      cl1=time.time()
8      alg( lst ,0, groesse -1)
9      cl2=time.time()
10     zeiten.append(cl2-cl1)
11   return min(zeiten), sum(zeiten)/float(100), max(zeiten)
```

In den Zeilen 7 und 9 werden die Zeiten vor und nach der Ausführung einer Sortierung gemessen. Die Zeiten der 100 Sortierungen werden in der Liste *zeiten* angesammelt; zurückgegeben wird schließlich das Minimum, das arithmetische Mittel und das Maximum der gemessenen Zeiten.

Schließlich kann mit einer Schleife über die Liste der zu untersuchenden Sortieralgorithmen die gewünschten Ergebnisse berechnet und ausgegeben werden.

for *qsAlg* **in** [*quicksortIP*, *quicksortRandomisiert*,*quicksortMedian*]:
  print *testAlg*(*qsAlg*)

Die gemessenen Zeiten ergaben sich bei mir:

	Min. Zeit	∅ Zeit	Max. Zeit	Abw.
`quicksortIP`	0.0710	0.0741	0.0794	11%
`quicksortRandomisiert`	0.1053	0.1080	0.1118	6%
`quicksortMedian`	0.1353	0.1370	0.1411	4.2%

Man sieht, wie deutlich sich die zusätzlichen Anweisungen der randomisierten Varianten in der Praxis bemerkbar machen. So benötigt die Median-Variante (obwohl sie ebenfalls eine asymptotische Laufzeit von $O(n \log n)$ besitzt) durschnittlich doppelt so lange um eine 10.000-elementige Liste zu sortieren wie die Basis-Variante. Man muss sich, bei der Verwendung einer randomisierten Variante von Quicksort, schon sehr genau darüber im Klaren sein, ob das in dem speziellen Fall auch Sinn macht.

Wie zu erwarten Nivellieren sich die Laufzeiten, je „randomisierter“ die Implementierung ist.

Aufgabe 2.9

Vergleichen Sie die Laufzeiten von *quicksortIter* und *quicksortIP* miteinander. Erklären Sie Ihre Beobachtungen.

Lösung

Folgende Tabelle zeigt einen Benchmark, der folgendermaßen erstellt wurde: Es wurden 10 zufällig erzeugte Listen der Größe 1000000 mit der iterativen Variante von Quicksort (*quicksortIter* , der rekursiven In-Place-Variante von Quicksort (*quicksortIP*) und Pythons interne Suchfunktion (*list* . *sort*() verglichen.

	Min. Zeit	∅ Zeit	Max. Zeit
`quicksortIter`	11.652	11.924	12.363
`quicksortIP`	11.262	11.3975	11.600
`list.sort()`	0.8547	1.1478	1.2800

Zwei erstaunliche Dinge fallen auf:

- Die iterative Variante ist etwa langsamer als die rekursive Variante. Dies hängt damit zusammen, dass der Python-Interpreter die Rekursion schon schlau genug optimiert; daher kann durch eine manuelle Eliminierung der Rekursion keine weitere Performance-Verbesserung erreicht werden.
- Pythons interne Sortierfunktion ist mehr als 10-mal schneller als die Quicksortimplementierung; das hängt mit folgender Tatsache zusammen: Quicksort benötigt –im Verleich zu anderen Sortieralgorithmen– verhältnismäßig wenige Vergleichsoperationen; es werden jedoch viele Vertauschungen vorgenommen; Vertauschungen sind aber Operationen auf dem Hauptspeicher, die im verglichen mit CPU-intern ausführbaren Operationen (wie Vergleiche und arithmetische Operationen) sehr langsam sind. Der Grund dafür reicht schon in die technische Informatik und hat mit dem immer größer werden Performance-Unterschied zwischen Speicher und CPU zusammen.

Aufgabe 2.13

(a) Implementieren Sie die Funktion *leftChild*, die als Argument eine Liste *lst* und einen Index i übergeben bekommt und, falls dieser existiert, den Wert des linken Kindes von $lst[i]$ zurückgibt; falls $lst[i]$ kein linkes Kind besitzt, soll *leftChild* den Wert *None* zurückliefern.

(b) Implementieren Sie die Funktion *rightChild*, die als Argument eine Liste *lst* und einen Index i übergeben bekommt und, falls dieser existiert, den Wert des rechten Kindes von $lst[i]$ zurückgibt; falls $lst[i]$ kein rechtes Kind besitzt, soll *rightChild* den Wert *None* zurückliefern.

(c) Implementieren Sie eine Funktion *father*, die als Argument eine Liste *lst* und einen Index i übergeben bekommt und den Wert des Vaters von $lst[i]$ zurückliefert.

Lösung

Am elegantesten fängt man eine *IndexError*-excpetion ab und liefert dann ein *None* zurück.

```
1 def leftChild( lst , i ):
2   try:
3     return lst[ i *2]
4   except IndexError:
5     return None
6
7 def rightChild( lst , i ):
8   try:
9     return lst[ i *2 + 1]
```

```
10    except IndexError:
11        return None
12
13  def father( lst , i ):
14    try:
15        return lst[ i/2 ]
16    except IndexError:
17        return None
```

Die in der *father*-Funktion verwendete Operation $i/2$ ist eine Integer-Division, die abrundet, was ja in der Tat auch notwendig ist, um den Vater eines bestimmten Knotens in einen als Liste repräsentieren Heap zu bestimmen (der Vater des Eintrags mit Index i ist ja der Eintrag mit Index $\lfloor i/2 \rfloor$.

Aufgabe 2.15

Wie arbeitet die Funktion *insert*, wenn das einzufügende Element x kleiner ist als die Wurzel des Heaps *lst* [1]? Spielen Sie den Algorithmus für diesen Fall durch und erklären Sie, warum er korrekt funktioniert.

Lösung

Wenn das einzufügende Element x kleiner ist als die Wurzel des Heaps, also als *lst* [1], so wird x durch wiederholten Tauschen bis an die Wurzel des Heaps hoch transportiert; im letzten Schritt ist befindet sich x dann in *lst* [1]. Der anschließende Vergleich der **while**-Schleife lautet dann:
while *lst*[1/2]>*lst*[1]
1/2 ergibt, da es sich um eine Integer-Division handelt, den Wert 0; es wird also der Vergleich *lst* [0]>*lst*[1] ausgeführt, was immer den Wert *False* liefert, da *None* per Definition immer kleiner ist als jeder andere Python-Wert. Die **while**-Schleife bricht also hier korrkterweise ab und x befindet sich nach Ausführung von *insertH* an der Wurzel des Heaps.

Aufgabe 2.18

Beantworten Sie folgende Fragen zu der in Listing **??** gezeigten Funktion *minHeapify*:

- In welchen Situationen gilt *len*(*nodes*)==3, in welchen Situationen gilt *len*(*nodes*)==2 und in welchen Situationen gilt *len*(*nodes*)==1?
- Können Sie sich eine Situation vorstellen, in der *len*(*nodes*)==0 gilt? Erklären Sie genau!

- Die Funktion *minHeapify* ist rekursiv definiert. Wo befindet sich der Rekursionsabbruch? Und: In welcher Hinsicht ist das Argument des rekursiven Aufrufs „kleiner“ als das entsprechende Argument in der aufrufenden Funktion.
  Denn, wie in Abschnitt **??** auf Seite **??** besprochen, müssen die rekursiven Aufrufe „kleinere“ (was auch immer „kleiner“ im Einzelnen bedeutet) Argumente besitzen als die aufrufende Funktion, um zu vermeiden, dass die Rekursion in einer Endlosschleife endet.

Lösung

(a) Immer dann wenn sich an Index i ein innerer Knoten befindet, der zwei Kinder hat, gilt $len(nodes)==3$. Wenn der Knoten jedoch nur ein einziges Kind hat (und dies ist bei einem Heap nur bei höchstens einem Knoten der Fall – es sei dem Leser überlassen, warum dies der Fall ist), so gilt $len(nodes)==2$, denn der r-Index wird in diesem Falle in der Listekomprehension durch die Bedingung $v \leq n$ ausgefiltert. Falls der Knoten an Index i ein Blatt ist, so gilt $len(nodes)==1$, denn in diesem Falle werden durch die Bedingung $v \leq n$ sowohl l als auch r ausgefiltert.

(b) Es gilt zu keinem Zeitpunkt, dass $len(nodes)==0$; die Listenkomprehension in Zeile 5 enthält zumindest *immer* den Wert $(heap[i], i)$, da zu keinem Zeitpunkt $i \leq n$ gilt.

(c) Der Rekursionsabbruch steckt implizit im fehlenden **else**-Fall der **if**-Anweisung in Zeile 8. Wenn nämlich gilt, dass $smallestIndex==i$, die Heap-Bedingung also erfüllt ist, erfolgt kein weiterer rekursiver Aufruf und die Funktion wird beednet.

Aufgabe 2.20

Eliminieren Sie die Listenkomprehension in Zeile 5 und deren Sortierung in Zeile 6 und verwenden Sie stattdessen **if**-Anweisungen mit entsprechenden Vergleichen um das kleinste der drei untersuchten Elemente zu bestimmen.

Lösung

Eine Version von *minHeapfy*, die statt der Listenkomprehension und deren Sortierung direkte Vergleiche verwendet:

```
1 def minHeapify(lst, i):
2   l = 2*i
3   r = l+1
4   n = len(lst)-1
5   if l≤n and lst[l]<lst[i]: smallest = l
6   else: smallest = i
7   if r≤n and lst[r]<lst[smallest]: smallest=r
8   if smallest≠i:
9     lst[i], lst[smallest] = lst[smallest], lst[i]
```

```
10    minHeapify(lst, smallest)
```

Aufgabe 2.22

Implementieren Sie – indem Sie sich an der Implementierung von *minHeapify* orientieren – die für Heapsort notwendige Funktion *minHeapify3*(i,n), die die übergebene Liste nur bis zu Index n als Heap betrachtet und versucht die Heapbedingung an Knoten i wiederherzustellen.

Lösung

Die Anpassung ist sehr einfach; statt n über die Länge der den Heap repräsentierenden Liste zu berechnen, wird n einfach als Parameter übergeben und wir erhalten so die folgende Implementierung vom *minHeapify3*:

```
1 def minHeapify3(lst,i,n):
2   l = 2*i
3   r = l+1
4   if l≤n and lst[l]<lst[i]: smallest = l
5   else: smallest = i
6   if r≤n and lst[r]<lst[smallest]: smallest=r
7   if smallest≠i:
8     lst[i], lst[smallest] = lst[smallest], lst[i]
9     minHeapify3(lst,smallest,n)
```

Aufgabe 2.23

Lassen Sie die Implementierungen von Quicksort und Heapsort um die Wette laufen – wer gewinnt? Versuchen Sie Ihre Beobachtungen zu erklären.

Lösung

Ich erhalte folgende Ergebnisse, wenn ich Heapsort und Quicksort beide auf 30 verschiedene zufällig erzeugte Listen der Länge 100000 anwenden:

	Min. Zeit	∅ Zeit	Max. Zeit
`quicksortIP`	0.8796	0.9179	0.9620
`heapSort`	2.2087	2.2810	2.3454

Die Quicksort-Implementierung ist also knapp 3-mal schneller als die Heapsort-Implementierung. Warum?

Aufgabe 2.24

Schreiben Sie eine möglichst performante Python-Funktion

smallestn(*lst* , *n*)

die die kleinesten n Elemente der Liste n zurückliefert.

Lösung

Man könnte die Aufgabe zwar einfach folgendermaßen lösen:

```
1 def smallestn( lst ,n):
2     lst . sort()
3     return lst[:n]
```

Dies ist jedoch nicht die performanteste Möglichkeit die kleinesten n Elemente der Liste *lst* zurückzuliefern; Laufzeit von *smallestn* wäre $O(l \log l)$, falls l die Länge der Liste *lst* ist. Es geht schneller mit Heaps:

```
1 import heapq
```

Aufgabe 3.1

Angenommen, ein (nehmen wir sehr recht schneller) Rechner kann ein Byte an Daten in 50 ns durchsuchen. Wie lange braucht der Rechner, um eine Datenbank einer Größe von 100 GB / 100 TB / 100 PB zu durchsuchen, wenn der Suchalgorithmus

(a) ... eine Laufzeit von $O(n)$ hat?

(b) ... eine Laufzeit von $O(\log(n))$ hat – nehmen Sie an, die Laufzeit wäre proportional zu $\log_2 n$ (was durchaus sinnvoll ist, denn meistens werden bei solchen Suchen binäre Suchbäume verwendet)?

Lösung

$50ns = 50 * 10^{-9}s$

(a) Verwendung der linearen Suche:
Laufzeit bei $100GB = 100 * 10^9$ Byte * $50 * 10^{-9}$ s/Byte = $100 * 50s = 5000s = 83min = 1$ Std 23 min
Laufzeit bei 100TB = $100 * 10^{12}$ Byte * $50 * 10^{-9}$ s/Byte = $5000 * 10^3$ s = $83000min = 1383Std > 57$ Tage
Laufzeit bei 100PB =$83000000min > 157$ Jahre

(b) Verwendung eines Suchbaums mit $O(\log(n))$ Laufzeit:
Laufzeit bei $100GB = 0,00182706ms = 1.827\mu s$
Laufzeit bei $100TB = 0,00232535ms = 2,325\mu s$
Laufzeit bei $100PB = 0,00282363ms = 2,823\mu s$

Aufgabe 3.5

Schreiben Sie die Funktion *search* iterativ.

Lösung

```
1  class BTree(object):
2    ...
3    def searchIter( self,s):
4      r=self
5      ende=False
6      val=None
7      while not ende:
8        if r==None: ende=True
9        elif s==r.key:
10         val=r.key
11         ende=True
12       elif s<r.key:
13         r=r.ltree
14       elif s>r.key:
15         r=r.rtree
16     return val
```

Listing 2: Suche in einem binären Suchbaum – iterativ

Aufgabe 3.6

Schreiben Sie eine Methode *BinTree.minEl*() und eine Methode *BinTree.maxEl*(), die effizient das maximale und das minimale Element in einem binären Suchbaum findet.

Lösung

Um das minimale Element in einem binären Suchbaum zu finden, müssen keinerlei Werte verglichen werden; aufgrund der besonderen Eigenschaften eines binären Suchbaums, genügt es einfach immer den Links-Verzweigungen nachzulaufen – gibt es keine Links-Verzweigung mehr, hat man das minimale Element erreicht. Das Finden des maximalen Elements geht analog.

```
1  class BTree(object):
2    ...
3    def minEl(self):
```

```
4       r = self
5       while r.ltree ≠ None:
6           r = r.ltree
7       return r
```

Listing 3: Effizientes Finden des minimalenElementes in einem binären Suchbaum

---

Aufgabe 3.10

Man kann ein destruktives Löschen unter Anderem unter Verwendung einer „Rückwärtsverzeigerung“ implementieren, d. h. unter Verwendung einer Möglichkeit, den Vaterknoten eines Knotens v anzusprechen.

Implementieren Sie diese Möglichkeit, indem Sie die Klasse *BTree* um ein Attribut *parent* erweitern. Man beachte, dass dies weitere Änderungen nach sich zieht: Die Methode *insert* muss etwa angepasst werden.

Lösung

Um die *parent*-Attribute gleich beim Erzeugen einer Klasseninstanz richtig zu setzten, muss die Konstruktorfunktion *__init__* der Klasse entsprechend angepasst werden:

```
1 class BTree(object):
2   def __init__(self, key, ltree=None, rtree=None, val=None):
3       self.ltree = ltree
4       if hasattr(ltree,'parent'): ltree.parent=self
5       self.rtree = rtree
6       if hasattr(rtree,'parent'): rtree.parent=self
7       self.key    = key
8       self.val    = val
9       self.parent = None
```

Listing 4: Erweiterung der Konstruktorfunktion der Klasse BTree um das Attribute *parent*

Pythons interne Funktion *hasattr* kann testen, ob ein bestimmtes Objekt ein Attribut besitzt; die beiden **if**-Abfragen in den Zeilen 4 und 6 stellen sicher, dass Kinderknoten (die entsprechend selbst ein Attribut *parent* haben müssen, für die also *hasattr*(*ltree* , `'parent'`) den Wert *True* liefert) den Rückzeiger auf den Vaterknoten erhalten.

Auch beim Einfügen in einen bestehenden binären Suchbaum, realisiert durch die Methode *insert*, muss jetzt zusätzlich darauf geachtet werden, dass die Rückzeiger auf die Vaterknoten korrekt gesetzt werden. In den Zeilen 5 und 10 im folgenden Listing werden diese Rückzeiger gesetzt.

```
1   def insert(self,x,val=None):
```

```
2     if x < self.key:
3       if self.ltree == None:
4         self.ltree = BTree(x)
5         self.ltree.parent = self
6       else: self.ltree.insert(x)
7     elif x > self.key:
8       if self.rtree == None:
9         self.rtree = BTree(x)
10        self.rtree.parent = self
11      else: self.rtree.insert(x)
```

Listing 5: Anpassung der Funktion *insert* um das neu hinzugefügte Attribut *parent* richtig zu setzen

Last but not least – und eigentlich unerwarteterweise: Die Implementierung von *BTree.deleteND* muss ebenfalls angepasst werden; dies liegt daran, dass bei der Konstruktion des Rückgabebaums mit Referenzen auf den alten Baum gearbeitet wird. Und in den Zeilen 10, 13 und 15 in Listing **??** baut *deleteND* mit diesen Refernzen einen neuen binären Suchbaum auf. Aufgrund der oben beschrieben Anpassungen der Konstruktorfunktion werden nun aber die *parent* Zeiger mit „umgebogen“, was (aufgrund der Tatsache, dass die Teilbäume keine neuen Kopien sondern lediglich Refernzen sind) zu unerwünschten Seiteneffekten auf den ursprünglichen binären Suchbaum führt – dessen *parent*-Zeiger werden dadurch nämlich umgebogen. Man müsste hier vorher neue Kopien der Teilbäume erzeugen; dies kann mittel *copy.deepcopy*() erfolgen.

In der Tat ist dies nur eine „Instanz“ eines allgemeineren Problems: die Speicherung redundanter Informationen (und die Rückverzeigerung mittels *parent* ist eine solche) zusammen mit der Referenz-Semantik verträgt sich schlecht mit der Implementierung nicht-destruktiver Updates.

Aufgabe 3.11

Implementieren Sie eine Methode *BTree.delete*(v), die auf destruktive Art und Weise einen Knoten mit Schlüsselwert v aus einem binären Suchbaum löscht.

Lösung

Wir beginnen mit einer rekursiven Lösung: soll ein Knoten geschlöscht werden, der noch (mindestens) einen Kindsknoten hat, so wird der Knoten – je nachdem welches Kind existiert – entweder mit dem maximalen Knoten des linken Teilbaums oder mit dem minimalen Knoten des rechten Teilbaums getauscht und anschließend rekursiv der getauschte Knoten gelöscht; dies geschieht in den Zeilen 7 und 12 in unten stehendem Listing. Ist der Knoten dagegen ein Blatt, so wird er einfach gelöscht (Zeile 14: *self.detach*()).

```
1   def delete( self,v):
```

```
2       if v==self.key:
3         if not self.rtree == None:
4           min = self.rtree.minEl()
5           self.key = min.key
6           self.val = min.val
7           self.rtree.delete(min.key) # Rekursion
8         elif not self.ltree == None:
9           max = self.ltree.maxEl()
10          self.key = max.key
11          self.val = max.val
12          self.ltree.delete(max.key) # Rekursion
13        else:
14          self.detach()
15      elif v<self.key and self.ltree≠None:
16          self.ltree.delete(v)
17      elif v>self.key and self.rtree≠None:
18          self.rtree.delete(v)
19
20    def detach(self):
21      if self.parent.rtree == self:
22        self.parent.rtree = None
23      elif self.parent.ltree == self:
24        self.parent.ltree = None
```

Listing 6: Destruktives Löschen – rekursive Lösung

An der folgenden iterativen Lösung wird noch deutlicher, dass eine destruktive Implementierung mehr lokale Deklarationen, mehr Abfragen und deutlich mehr Komplexität in der Implementierung nach sich zieht als die vorgestellte nicht-destruktive Version; in der Tat ist es ja auch eine der Stärken der funktionalen Programmierung komplexe Algorithmen kompakt und verständlich formulieren zu können.

Folgendes Listing zeigt also die Implementierung des destruktiven Löschens in einem binären Suchbaum, realisiert als Methode der Klasse *BTree*.

```
1 class BTree(object):
2   ...
3   def delete(self,v):
4     if self.key == v:
5       if self.ltree == None or self.rtree == None:
6         z = self
7       else:
8         z = self.rtree.minEl()
9       # z loeschen + evtl hochhieven...
10      x = None
11      if z.rtree ≠None:
```

```
12            x = z.rtree
13          elif z.ltree ≠ None:
14            x = z.ltree
15          u = z.parent
16          if u.ltree == z:
17            u.ltree = x
18          else:
19            u.rtree = x
20          if z≠self: # z muss nach oben kopiert werden
21            self.key = z.key
22            self.val = z.val
23        elif v < self.key:
24          self.ltree.delete(v)
25        elif v > self.key:
26          self.rtree.delete(v)
```

Listing 7: Destruktives Löschen in einem binären Suchbaum

Übrigens fehlt noch – was obiges Listing noch weiter aufblähen würde – eine Abfrage, ob sich der zu löschende Knoten und der z-Knoten nicht an der Wurzel befinden – zwar ein seltener Fall, müsste aber eigentlich abgefangen werden.

Aufgabe 3.13

Gegeben seien die Schlüssel $51, 86, 19, 57, 5, 93, 8, 9, 29, 77$.

(a) Welche Höhe hat der Baum, wenn die Schlüssel in der oben angegebenen Reihenfolge in einen anfänglich leeren Baum eingefügt werden?

(b) Finden Sie eine Einfügereihenfolge, bei der ein Baum der Höhe 9 entsteht.

(c) Finden Sie eine Einfügereihenfolge, bei der ein Baum minimaler Höhe entsteht.

Lösung

(a) Der so entstandene Baum hat eine Höhe von 4

(b) Werden die Zahlen in aufwärts oder abwärts sortierter Reihenfolge eingefügt, so entsteht ein entarteter Baum mit Höhe 9.

(c) Vertauscht man die Einfügereihenfolge von 5 und 8 so ensteht ein Baum mit Höhe 3; dies ist auch gleichzeitig die minimal mögliche Höhe eine Binärbaumes der 10 Werte enthält ($\lceil \log_2 10 \rceil - 1 = 3$)

Aufgabe 3.15

Implementieren Sie ...

(a) ... eine Methode *_simpleRight* der Klasse *AVLTree*, die eine einfache Rechtsrotation realisiert.

(b) ... eine Methode *_doubleRight* der Klasse *AVLTree*, die eine Doppel-Rechts-Rotation realisiert.

Lösung

(a) Eine Methode *_simpleRight* der Klasse *AVLTree*, die eine einfache Rechtsrotation realisiert:

```
1  def _simpleRight( self ):
2      a     = self ; b = self. ltree
3      t1    = b. ltree
4      t23   = b. rtree
5      t4    = a. rtree
6      newR = AVLTree(a.key, t23, t4, a.val)
7      self .key = b.key ;  self . ltree  = t1 ;  self . rtree  = newR ; self. val = b.val
```

(b) Eine Methode *_doubleRight* der Klasse *AVLTree*, die eine Doppel-Rechts-Rotation realisiert.

```
1  def _doubleRight( self ):
2      a = self ; b = self. ltree  ; c = self. ltree . rtree
3      t1 = b. ltree
4      t2 = c. ltree
5      t3 = c. rtree
6      t4 = a. rtree
7      newL = AVLTree(b.key, t1, t2, b. val)
8      newR = AVLTree(a.key, t3, t4, a.val)
9      self .key = c.key ;  self . ltree  = newL ; self. rtree  = newR ; self. val = c. val
```

Aufgabe 3.16

Um wie viel kann sich die Länge des längsten Pfades mit der Länge des kürzesten Pfades (von der Wurzel zu einem Blatt) eines AVL-Baums höchstens unterscheiden?

Lösung

Der maximle Längenunterschied zwischen Pfaden wächst linear mit der Höhe des Baumes, d. h. bei einem Baum der Höhe n ist der maximale Pfadlängenunterschied genau

n.

Aufgabe 3.17

(a) Wie hoch wäre ein (fast) vollständiger binärer Suchbaum, der 300000 Elemente enthält?

(b) Wie hoch könnte ein Rot-Schwarz-Baum maximal sein, der 300000 Elemente enthält?

Lösung

(a) Die minimale Höhe eine binären Suchbaums der 300000 Elemente enthält ist $\lceil \log_2 300000 \rceil = 19$.

(b) Damit ist die Höhe des erwähten rot-schwarz Baumes also um $7/19 \approx 36\%$ größer als die Höhe eine optimalen binären Suchbaums

---

Aufgabe 3.18

Schreiben Sie eine Methode *RBTree.inv1Verletzt*, die testet, ob es einen Knoten im Rot-Schwarz-Baum gibt, für den die Invariante 1 verletzt ist, d. h. ob es einen roten Knoten gibt, dessen Vorgänger ebenfalls ein roter Knoten ist. Ein Aufruf von *inv1Verletzt* soll genau dann *True* zurückliefern, wenn die Invariante 1 für mindestens einen Knoten verletzt ist.

Lösung

Die Methode, die einen rot-schwarz Baum darauf hin testet, ob einer seiner Knoten die Invariante 1 verletzt, kann folgenderaßen implementiert werden:

```
1 class RBTree(object):
2   ...
3   def inv1Verletzt( self ):
4     "-> True <=> Invariante 1 verletzt (d.h. zwei Rot-Knoten hintereinander)"
5     if self.l==None: l = False
6     else: l = self.l.inv1Verletzt() or ( self.c==self.l.c==RED)
7     if self.r==None: r = False
8     else: r = self.r.inv1Verletzt() or ( self.c==self.r.c==RED)
9     return l or r
```

Aufgabe 3.19

Schreiben Sie eine Methode, die überprüft, ob die Invariante 2 verletzt ist.

(a) Schreiben Sie hierfür zunächst eine Methode *RBTree.anzSchwarzKnoten*, die ein Tupel (x, y) zurückliefern soll, wobei in x die minimale Anzahl schwarzer Knoten auf einem Pfad von der Wurzel zu einem Blatt und in y die maximale Anzahl schwarzer Knoten auf einem Pfad von der Wurzel zu einem Blatt zurückgegeben werden soll.

(b) Schreiben Sie nun eine Methode *RBTree.inv2Verletzt*, die genau dann *True* zurückliefern soll, wenn die Invariante 2 für den entsprechenden Rot-Schwarz-Baum verletzt ist.

Lösung

(a) Die Methode, die die minimale und maximale Anzahl von Schwarz-Knoten auf einem Pfad von der Wurzel zu einem Blatt zurückliefert kann folgendermaßen implementiert werden:

```
1  class RBTree(object):
2      ...
3    def anzSchwarzKnoten(self):
4      if self.l==None: l=(0,0)
5      else: l = self.l.anzSchwarzKnoten()
6      if self.r==None: r=(0,0)
7      else: r = self.r.anzSchwarzKnoten()
8      if self.color == BLACK:
9        return (1 + min(l[0],r[0]), 1 + max(l[1],r[1]))
10     else:
11       return (min(l[0],r[0]), max(l[0], r[0]))
```

(b) Die Methode, die nun testet, ob die Invariante 2 für einen gegebenen Baum verletzt ist, kann folgendermaßen implementiert werden:

```
1  class RBTree(object):
2      ...
3   def inv2Verletzt( self ):
4      (minS, maxS) = self.anzSchwarzKnoten()
5      return minS ≠maxS
```

Die Methode *RBTree.inv2Verletzt* liefert nun genau dann *True* zurück, wenn die Invariante 2 für den rot-schwarz Baum verletzt ist.

Aufgabe 3.20

Vergleichen Sie die Performance des Python-internen *dict*-Typs mit der vorgestellten Implementierung von rot-schwarz Bäumen folgendermaßen:

(a) Fügen sie 1 Mio zufällige Zahlen aus der Menge $\{1, \ldots 10Mio\}$ jeweils in einen Python-*dict* und in einen Rot-Schwarz-Baum ein, messen sie mittels *time*() jeweils die verbrauchte Zeit und vergleichen sie.

(b) Führen sie nun 1 Mio Suchdurchgänge auf die in der vorigen Teilaufgabe erstellten Werte aus, und messen sie wiederum mittels *timeit* die verbrauchte Zeit und vergleichen sie.

Lösung

Ein direkter Performance-Vergleich brachte die folgenden Werte allerdings bei einer Anzahl von 10^5 einzufügenden Elementen:

`RBTree` Einfügen:	23.138
`dict` Einfügen:	0.034
`RBTree` Suchen:	2.173
`dict` Suchen:	0.0327

Als Fazit: Das Rebalancieren scheint wohl der größte Flaschenhals. Warum der *dict*-Typ jedoch so viel performanter sein kann als ein rot-schwarz Baum erfahren wir im Kapitel über Hashing.

Aufgabe 3.22

Schreiben Sie mittels einer Listenkomprehension die in Listing **??** gezeigte Funktion *hashStrSimple* als Einzeiler.

Lösung

Folgendermaßen kann die in Listing **??** gezeigte Implementierung von *hashStringSimple* als Einzeler realisiert werden:

```
1 def hashStrSimple1(s,p):
2   return sum( [ord(s[len(s) -1 -i])≪(8 ∗i) for i in range(len(s)) ]) % p
```

Aufgabe 3.23

Implementieren Sie das Horner-Schema in einer Schleife – anstatt, wie in Listing **??** die Python-Funktion *reduce* zu verwenden.

Lösung

Folgendermaßen kann das Horner-Schema in einer Schleife implementiert werden:

```
1 def hornerLoop(l,b):
2   v=0
3   for i in range(0,len(l)):
4     v = v<<b + l[i]
5   return v
```

Aufgabe 3.24

Verwenden Sie, statt *reduce* und *map*, eine Schleife, um die in Listing **??** gezeigte Funktion *hashStr* zu implementieren.

Lösung

Folgendermaßen kann die mittels des Horner-Schemas berechnete Hash-Funktion durch eine Schleife implementiert werden.

```
1 def hashStr(s,p):
2   v=0
3   for i in range(0,len(s)):
4     v = (v<<8 + ord(s[i]))
5     print v
6   return v% p
```

Aufgabe 3.25

Ganz offensichtlich ist nicht, welche der Funktionen *horner* und *horner2* tatsächlich schneller ist – auf der einen Seite vermeidet *horner2* die Entstehung großer Zahlen als Zwischenergebnisse; andererseits werden in *horner2* aber auch sehr viel mehr Operationen (nämlich Modulo-Operationen) ausgeführt als in *horner*.

Ermitteln Sie empirisch, welcher der beiden Faktoren bei der Laufziet stärker ins Gewicht fällt. Vergleichen Sie die Laufzeiten der beiden Funktionen *horner* und *horner2* mit Listen der Länge 100, die Zufallszahlen zwischen 0 und 7 enthalten, mit Parameter $b = 3$ und einer dreistelligen Primzahl. Verwenden Sie zur Zeitmessung Pythons *timeit*-Modul.

Lösung

Die Ausführung des folgenden Programmfragments (zusammen mit den Implementierungen der beiden Funktionen *horner* und *horner2*)

```
from random import randint
from timeit import Timer
l = [randint(0,7) for _ in range(1000)]
t1 = Timer('horner(l,3,163)', 'from __main__ import horner, l')
t2 = Timer('horner2(l,3,163)', 'from __main__ import horner2, l')
print "Laufzeit horner: ",t1.timeit(number=10000)
print "Laufzeit horner2: ",t2.timeit(number=10000)
```

liefert:

```
Laufzeit horner: 0.036484956741333 01
Laufzeit horner2: 0.23898601531982422
```

Die vielen Modulo-Operationen scheinen also wesentlich weniger ins Gewicht zu fallen als das Rechnen mit großen Zahlen.

Aufgabe 3.28

Definieren Sie sich eine Instanz der Methode *__str__*, um sich die für den Benutzer relevanten Daten von Objekten vom Typ *Entry* anzeigen zu lassen.

Lösung

Folgendermaßen kann eine Definition der Methode *__str__* aussehen, die die für den Benutzer relevanten Daten, also den Schlüssel und den dazugehörigen Wert anzeigen.

```
class Entry(object):
  ...
  def __str__(self):
    return "<Entry: %s:%s>" % (self.key, self.value)
```

Aufgabe 3.30

Angenommen, unsere Hash-Tabelle hat eine Größe von 2^{20} und enthält 900 000 Werte. Angenommen, wir würden *keine* zweite Hash-Funktion verwenden wollen, sondern stattdessen einfaches Hashing.

(a) Passen Sie hierfür die **while**-Schleife in Zeile 12 aus Listing **??** so an, dass sie den Schlüssel *key* unter der Annahme sucht, dass die Hash-Tabelle mit linearem Hashing befüllt wurde.

(b) Wie oft müsste die so implementierte **while**-Schleife im Durchschnitt durchlaufen werden, bis ein in der Hash-Tabelle befindlicher Schlüssel gefunden wird?

(c) Wie oft müsste die so implementierte **while**-Schleife im Durchschnitt durchlaufen werden, bis die *_lookup*-Funktion „merkt“, dass der zu suchende Schlüssel *key* sich nicht in der Hash-Tabelle befindet?

Lösung

(a) Man kann die **while**-Schleife folgendermaßen umprogrammieren um die Suche bei einem linearen Hashing zu realisieren:

```
1  ...
2  while True:
3    i = (i+1) & self.mask
4    entry = self.table[i]
5    if entry.key==None or entry.key==key:
6      return entry
```

(b) Nach der Formel

$$\frac{1}{2} + \frac{1}{2}(1-\beta)^2$$

erhält man mit $\beta = 900000/2^{20} = 0.8583$ für die Anzahl der Suchschritte bei einer erfolglosen Suche: 25 Schritte.

(c) Nach der Formel

$$\frac{1}{2} + \frac{1}{2}(1-\beta)$$

erhält man mit $\beta = 0.8583$ für die Anzahl der Suchschritte bei einer erfolgreichen Suche: 4 Schritte.

Aufgabe 3.32

Die Selektion der i niederwertigsten Bits entspricht eigentlich der Operation „% 2^i“. Dies widerspricht eigentlich der Empfehlung aus Abschnitt **??**, man solle als Hash-Funktion „% p“ mit p als Primzahl verwenden. Argumentieren Sie, warum dies hier durchaus sinnvoll ist.

Lösung

Die Empfehlung eine Hash-Funktion $\mod p$ mit p, Primzahl, zu Verwenden bezieht sich auf das Ziel, die Bits möglichst gut durchzumischen - und genau diese Bit-Durchmischung ist optimal, wenn p eine Primzahl ist.

Die $\mod 2^i$-Operation aus der *OurDict*-Implementierung hat aber gar keine Bit-durchmischung zum Ziel; die Bits wurden schon vorher durch die *hashStr* durchmischt; die $\mod 2^i$-Operation ist alleine dazu da, die relevanten Bits zu selektieren und den Hash-Wert auf das Intervall $\{0, \ldots 2^i - 1\}$ abzubilden.

Aufgabe 3.34

Programmieren Sie die Funktion *_insert_init* .

Lösung

```
1  class OurDict(object):
2   ...
3  def _insert_init ( self ,entry):
4     i = entry.hash & self .mask
5     new_entry = self. table [ i ]
6     perturb = entry.hash
7     while new_entry.key ≠None:
8        i = (i≪2) + i + perturb + 1
9        new_entry = self. table [ i & self .mask]
10       perturb = perturb ≫ PERTURB_SHIFT
11    new_entry.key = entry.key
12    new_entry.value = entry.value
13    new_entry.hash = entry.hash
14     self .used += 1
```

Im Gegensatz zur *_insert*-Funktion wird hier der Hash-Wert nicht erneut berechnet, sondern der bestehende genommen und mit ihm die Position in der Tabelle bestimmt, an der er eingefügt werden soll. Anders als *_insert* wird hier kein Aufruf an *_lookup* verwendet, hauptsächlich aus Performance-Gründen, weil in man diesen Performance-„Flaschenhals“ so wenig wie möglich Overhead haben will.

Aufgabe 3.35

Definieren Sie für den *OurDict*-Typ die Methode *__getitem__*, mit deren Hilfe man einfach den Wert eines Schlüssels durch Indizierung erhält.

Lösung

```
1  class OurDict(object):
2   ...
3  def __getitem__( self , key):
4     value = self._lookup(key). value
5     if value==None:
6       raise KeyError("Schluessel existiert nicht")
```

```
7       return value
```

Hier wird einfach durch Aufruf der *_lookup*-Funktion der entsprechende Eintrag in der Hash-Tabelle gesuchgt und der Wert zurückgeliefert.

Aufgabe 3.36

Implementierung Sie für den *OurDict*-Typ eine Möglichkeit, Elemente zu löschen, d. h. definieren Sie eine Instanz der Methode *__delitem__*. Auf was müssen Sie dabei besonders achten?

Lösung

```
1   def __delitem__(self, key):
2           entry = self._lookup(key)
3           if entry.value == None:
4               raise KeyError("Schluessel nicht vorhanden")
5           self._del(entry)
6
7   def _del(self, entry):
8           """
9           Mark an entry as free with the dummy key.
10          """
11          entry.key = dummy
12          entry.value = None
13          self.used -= 1
```

Die in der Aufgabenstellung erwähnte Schwierigkeit besteht darin, dass ein Eintrag, der gelöscht wurde, besonders markiert werden muss; würde diese Markierung nicht erfolgen, so könnten bestimmte Schlüssel, die eigentlich derselbsen Tabellenposition zugeordnet wären, durch frühere Kollisionen aber „weggehasht“ wurden, nicht mehr gefunden werden.

Es bleibt dem Leser überlassen alle weiteren Anpassungen (z.B: in der Funktion *_lookup*) vorzunehmen, die diesen *dummy*-Wert berücksichtigen. Desweiteren darf sich der Leser selbst überlegen, wie man *dummy* definieren könnte – es gibt vielen Möglichkeiten hierfür.

Aufgabe 3.37

Warum ist es nicht sinnvoll, dem Python-Typ *list* eine Implementierung der *__hash__*-Methode zu geben? In anderen Worten: warum können Listen nicht als Index eines

dict-Objekt dienen? Was könnte schief gehen, wenn man auf ein Element mittels eine Liste zugreifen möchte, wie etwa in folgendem Beispiel:

```
>>> lst = [1,2,3]
>>> d = { lst:14, 'Hugo':991 }
```

Lösung

Der Grund, dass es in Python nicht möglich ist, Dictionaries mittels Listen zu indizieren, liegt daran, dass Listen veränderlich sind. Nehmen wir an, wir würden eine gegebene Liste

```
>>> lst=[1,2,3]
```

dazu verwenden, einen Wert eines Dictionaries zu indizieren. Um die Positionen in der dem Dictionary zugrunde liegenden Hashtabelle zu bestimmen, müsste eine *__hash__*-Methode auf die Liste angewendet werden, die der Liste einen Integer-Wert zuweist.

Wird jedoch der Wert von *lst* im Laufe des Programms verändert, so hätte dies zur Folge, dass auch *lst.__hash__* einen anderen Wert zurückliefern würde und dies wiederum würde bedeuten, dass *adict[lst]* einen anderen, bzw. einen undefinerten Wert zurückliefern würde. So etwas will man im Allgemeinen vermeiden; deshalb ist es sinnvoll, nur Werte als Index in einem Dictionary zu zulassen, die unveränderlich sind, mit denen also die eben beschriebenen „überraschenden“ Effekte nicht auftreten können.

Aufgabe 3.39

(a) Erklären Sie, warum folgender Methode der Klasse *BloomFilter nicht* geeignet ist, ein Element aus dem Bloomfilter zu entfernen:

```
def delete(self,x):
  for i in range(0, self.k):  self.A[self.h[i](x) % self.m] = False
```

(b) Schreiben Sie die Methode *delete* so um, dass sie ebenfalls das Element x löscht, jedoch mit möglichst wenig „Seiteneffekten“.

(c) Warum ist selbst die in der letzten Teilaufgabe programmierte Lösch-Funktion in vielen Fällen nicht sinnvoll?

Lösung

(a) Der vorgestellte Algorithmus setzt alle Bits, die durch das zu löschende Element x mittels der Hash-Funktionen bestimmt sind, auf *False*. Dies löscht zwar das Element, hat jedoch den Seiteneffekt, dass alle im Bloomfilter enthaltenen Elemente y, die auf eine der gelöschten Positionen abgebildet werden, gelöscht sind. Genauer: Es werden

alle y gelöscht, für die gilt, dass

$$\{\ self.array[\ self.h[i](x)\ \%\ self.m \mid i \in \{0,\ldots,\ self.k-1\}\ \}\ \cap$$
$$\{\ self.array[\ self.h[i](y)\ \%\ self.m \mid i \in \{0,\ldots,\ self.k-1\}\ \}\ \neq\ \emptyset$$

(b) Eine alternative Implementierung mit etwas weniger Seiteneffekten:

```
1 class BloomFilter(object):
2   ...
3   def delete(self,x):  self.array[self.h[0](x) % self.m] = False
```

(c) Das Entfernen nur einer *True*-Position löscht zwar das Element, aber auch dieses „vorsichtige“ Entfernen kann Seiteneffekte haben: Jedes Element y, für das gilt:

$$\exists i \in \{0,\ldots,\ self.k-1\} : (\ self.h[i](y)\ \%\ self.m) = (\ self.h[0](x)\ \%\ self.m)$$

wird auch durch dieses „vorsichtige“ Löschen mit gelöscht. Die Folge ist, dass es auch in Folge solcher „vorsichtiger“ Lösch-Aktionen die Möglichkeit für falsch-negative Aussagen gibt; während man zwar (eine geringe Anzahl) falsch-positiver Membership-Tests bei Bloomfiltern hinnimmt, will man jedoch grundsätzlich die Möglichkeit falsch-negativer Membership-Tests ausschließen. Dies verbietet auch das „vorsichtige“ Löschen.

Aufgabe 3.41

Eine bessere Möglichkeit (als die in Aufgabe **??** vorgestellte), eine Lösch-Funktion zu implementieren, ist die Verwendung eines sog. *Countingfilters*. Ein Countingfilter ist ein Bloomfilter, dessen Einträge keine Bitwerte (d. h. *True* oder *False*) sind, sondern Zähler. Anfänglich sind alle Einträge 0; mit jeder Einfüge-Operation werden die durch die Hash-Funktion bestimmten Einträge des Bloomfilter-Arrays jeweils um Eins erhöht.

(a) Implementieren Sie, angelehnt an die in Listing **??** gezeigte Implementierung der Klasse *BloomFilter*, eine Klasse *CountingFilter*, die einen Countingfilter implementiert. Implementieren Sie eine Methode *insert*, die ein Element einfügt, und eine Methode *elem*, die testet, ob ein Element in dem Bloomfilter enthalten ist.

(b) Implementieren Sie eine Methode *delete*, die ein Element in einem Bloomfilter löscht.

Lösung

(a) Die Implementierung der Klasse *CountingFilter*

```
1 class CountingFilter( object ):
2   def __init__ ( self ,  bloomHashs, m):
```

```
3       self.k = len(bloomHashs)
4       self.h = bloomHashs
5       self.array = [0]*m
6       self.m = m
7
8     def insert(self,x):
9       for i in range(0, self.k):
10        self.array[self.h[i](x) % self.m] += 1
11
12    def elem(self,x):
13      return 0 not in [self.array[self.h[i](x) % self.m]
14                                              for i in range(0, self.k)]
```

(b) Die Implementierung der Lösch-Funktion.

```
1  class CountingFilter(object):
2    ...
3    def delete(x):
4      if self.elem(x):
5        for i in range(0, self.k):
6          self.array[self.h[i](x) % self.m] -= 1
```

Aufgabe 3.43

Beantworten Sie die folgenden Fragen:

(a) Wie viele Hash-Funktionen sollte man verwenden, bei einem Bloomfilter der Größe 1 MBit, das etwa 100000 Elemente speichern soll?

(b) Wie viele Bits pro gespeichertem Eintrag werden von einem Bloomfilter benötigt, dessen Fehlerrate höchstens bei 1% liegen soll?

(c) Wie viele Bits pro gespeichertem Eintrag werden von einem Bloomfilter benötigt, dessen Fehlerrate höchstens bei 0.1% liegen soll?

Lösung

(a) Hier gilt $m = 1MBit = 10^6 Bit$ und $n = 100000 = 10^5$. Der optimale Wert von k ergibt sich also zu

$$k = \ln(2) \cdot 10^6/10^5 = \ln(2) \cdot 10 = 6.931...$$

In diesem Fall wäre es also am günstigsten 6 oder 7 Hash-Funktionen zu verwenden – man würde wohl eher 6 (oder etwas weniger) wählen, um die Rechenzeit zu minimieren.

(b) Wollen wir eine Fehlerrate von höchstens 1%, so gilt

$$m \geq n \log_2(100) \Leftrightarrow m/n \geq \log_2(100) = 6.64...$$

Der Belegungsfaktor $m/n = 6.64...$ bedeutet, dass pro Eintrag etwa 6.64... Einträge, also 6.64... Bits benötigt werden.

(c) Wollen wir eine Fehlerrate von höchstens 0.1%, so gilt

$$m \geq n \log_2(10) \Leftrightarrow m/n \geq \log_2(1000) = 9.96...$$

Der Belegungsfaktor $m/n = 9.96...$ bedeutet, dass pro Eintrag etwa 9.96... Einträge, also 9.96... Bits benötigt werden.

Aufgabe 3.46

Implementieren Sie die Funktion *__str__* , so dass Skip-Listen folgendermaßen ausgegeben werden:

```
>>> print skiplist
>>> [ (30|1), (33|4), (40|3), (77|1), (98|1), (109|1), (193|3) ]
```

Ausgegeben werden soll also der Schlüssel jedes Elements zusammen mit der Höhe des Elements.

Lösung

Eine passende *__str__*-Methode für SkipListen:

```
class SkipList( object ):

  def __str__ ( self ):
    retStr = '[ '
    x = self.head
    while x.ptrs[0] is not self. tail :
      x = x.ptrs[0]
      retStr += '(' + str(x.key) + '|' + str(len(x.ptrs) -1) + '), '
    return retStr[: -2] + ' ]'
```

Die Anweisung *retStr*[: −2] in der **return**-Anweisung dient einfach dazu, das Komma und das Leerzeichen, das dem letzten Skip-Listen-Element folgt, zu entfernen.

Aufgabe 3.47

(a) Schreiben Sie eine Methode *keys*(), die eine Liste der in der Skip-Liste gespeicherten Schlüsselwerte zurückliefert.

(b) Schreiben Sie eine Methode *vals*(), die eine Liste der in der Skip-Liste gespeicherten Werte zurückliefert.

Lösung

Die Implementierung der Methoden *keys*() und *vals*() die eine Liste der in der jeweiligen Skip-Liste gespeicherten Schlüsselwerte bzw. eine Liste der Werte zurückliefern:

```
class SkipList( object ):
  ...
  def keys( self ):
    x = self.head
    k = []
    while x.ptrs[0] is not self. tail :
      k. append(x.key)
      x=x.ptrs[0]
    return k

  def vals( self ):
    x = self.head
    v = []
    while x.ptrs[0] is not self. tail :
      v. append(x.val)
      x = x.ptrs[0]
    return v
```

Aufgabe 3.48

Oft wird eine effiziente Bestimmung der Länge einer Skip-Liste benötigt. Erweitern Sie die Klasse *SkipList* um ein Attribut *length*, passen Sie entsprechend die Methoden *insert* und *delete* an und geben Sie eine Implementierung der Methode *__len__* an, so dass die *len*-Funktion auf Skip-Listen anwendbar ist.

Lösung

Hinzufügen einer effizienten Längenbestimmung einer Skipliste.

Hier die leichten Modifikationen, der *__init__* - und der *insert*-Methode, die einfach dann, wenn ein zusätzliches Element der Skip-Liste hinzugefügt werden soll das *self . length*-Attribut um Eins erhöht wird.

```
class SkipList( object ):
  def __init ( self ):
    ...
    self . length = 0
```

```
def insert( self ,key, val ):
  updatePtrs = [self.head] *(MaxHeight + 1)
  x = self.head
  for i in range( self.height, -1, -1):
    while x.ptrs[i].key < key: x = x.ptrs[i]
    updatePtrs[i] = x
  x = x.ptrs[0]
  if x.key==key: x.val = val
  else: #wirklich einfuegen
    self.length += 1
    ...
```

Die Modifikation der *delete*-Methode geht entsprechend.

Die Implementierung der *__len__*-Funktion ist dann offensichtlich:

```
def __len__( self ):
  return self.length
```

Aufgabe 3.51

Zeichnen Sie einen Trie, der die Schlüssel `gans`, `ganz`, `galle`, `leber`, `lesen`, `lesezeichen`, `zeichnen`, `zeilenweise`, `adam`, `aaron` speichert und beantworten Sie die folgenden Fragen:

(a) Wie viele Schritte benötigt eine Suche in diesem Trie minimal?

(b) Wie viele Schritte benötigt eine Suche in diesem Trie maximal?

Lösung

(a) die minimale Anzahl der Character-Vergleiche für eine Suche in diesem konkreten Trie beträgt 1 – beispielsweise für die Suche nach einem Wort, das mit `'b'` beginnt.

(b) Die maximale Anzahl der Schritte bei der Suche in einem Trie entspricht der Länge des längsten im Trie enthaltenen Schlüssels. In diesem konkreten Fall wäre dies *len*(`'zeilenweise'`)=11; für eine beliebige Suche sind also höchstens 11 Charakter-Vergleiche notwendig.

Aufgabe 3.52

Beantworten Sie die folgenden Fragen:

(a) Wie viele Character-Vergleiche benötigt eine Suche in einem Trie höchstens, der 1 Mio verschiedene Schlüsselwerte mit einer Länge von höchstens 14 enthält?

(b) Wie viele Character-Vergleiche benötigt eine Suche in einem binären ausgeglichenen Suchbaum, der 1 Mio verschiedene Schlüsselwerte mit einer Länge von höchstens 14 enthält?

Lösung

Hier zeigt sich sehr deutlich die Überlegenheit der Tries gegenüber herkömmlichen binären Suchbäumen, wenn es darum geht, große zusammengesetzte Werte zu speichern.

(a) Solche ein Trie, der 1 Mio verschiedenen Wörter der Länge ≤ 14 speichert, benötigt für eine Suche höchstens 14 Character-Vergleiche.

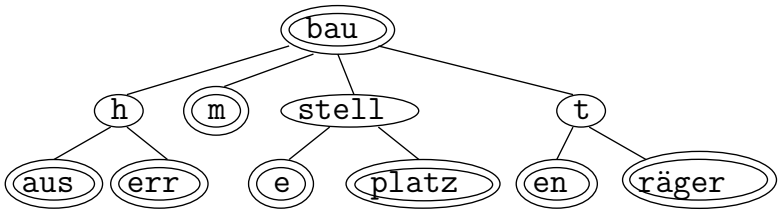
(b) Ein Binärbaum, der 1 Mio verschiedene Wörter der Länge ≤ 14 speichert, benötigt für eine Suche höchstens $14 \cdot \lceil \log_2 10^6 \rceil = 14 \cdot 21 = 294$ Vergleiche.

Aufgabe 3.54

Fügen Sie in den Patricia-Trie aus Abbildung **??** die Schlüsselwerte `baustellplatz` und `bauträger` ein.

Lösung

Durch dieses Einfügen ergibt sich der folgende Patricia-Baum:



Aufgabe 4.1

Implementieren Sie die Vereinigungs-Operation *mergeHeaps*, die zwei binäre Heaps miteinander vereinigt. Welche Laufzeit hat ihre Implementierung?

Lösung

Die folgende Funktion *mergeHeaps* vereinigt zwei Heaps:

```
1 def merge(h1,h2):
2   hMerged = h1 + h2[1:]
3   buildHeap(hMerged)
4   return hMerged
```

Laufzeit: $O(len(h1) + len(h2))$

Aufgabe 4.2

Wie viele Knoten hat ein Binomial-Baum der Ordnung k?

(a) Schreiben Sie eine rekursive Python-Funktion *anzKnotenBinomial*(k), die die Anzahl der Knoten eines Binomial-Baums der Ordnung k zurückliefert; diese Funktion sollte sich an der induktiven Definition eines Binomial-Baums orientieren.

(b) Zeigen Sie mit Hilfe der vollständigen Induktion, dass ein Binomial-Baum der Ordnung k genau 2^k Elemente enthält.

Lösung

(a) Das folgende (ineffiziente) Progamm berechnet die Anzahl der Knoten eines Binomialbaums der Ordnung k:

```
1 def anzKnotenBinomial(k):
2   if k==0: return 1
3   else:
4     return 1 + sum( [anzKnotenBinomial(i) for i in range(k)] )
```

(b) Induktionsanfang ($k = 0$): Ein Binomialbaum der Ordnung „0“ hat lt. Definition genau $1 = 2^0$ Knoten.
Induktionsschritt ($k \to k + 1$): Angenommen, ein Binomialbaum der Ordnung k habe 2^k Knoten. Ein Binomialbaum der Ordnung $k + 1$ besteht nun aus einem Wurzelknoten und aus einem Binomialbaum der Ordnung k (mit 2^k Knoten), einem Binomialbaum der Ordnung $k - 1$ (mit 2^{k-1} Knoten), usw. Insgesamt hat also ein Binomialbaum der Ordnung $k + 1$ genau

$$2^k + 2^{k-1} + \ldots + 2^0 = 1 + \sum_{i=0}^{k} 2^i = 1 + (2^{k+1} - 1) = 2^{k+1}$$

Knoten.

Aufgabe 4.3

Implementieren Sie eine Python-Funktion *isBinomial*(bt), die genau dann „*True*“ zurückliefert, wenn das Argument bt ein gültiger Binomial-Baum ist.

Lösung

Folgende Funktionen testen ihr Argument darauf hin ab, ob sie von der Struktur her gültige Binomialbäume sind.

```
1 def isBinomTree(bt):
2   k = len(bt[1])
3   return isBinomTreeK(k,bt)
4
5 def isBinomTreeK(k,bt):
6   if k==0:
7     return len(bt[1])==0
8   else:
9     return len(bt[1])==k and all(isBinomTreeK(i,bt[1][k-1-i]) for i in range(k-1,-1,-1)) and all(
```

Aufgabe 4.6

Implementieren Sie eine Python-Funktion *isBinHeap*(*bh*), die genau dann „*True*“ zurückliefert, wenn *bh* ein gültiger Binomial-Heap ist.

Lösung

Folgende Python-Funktion *isBinHeap*(*bh*), liefert genau dann „*True*“ zurück, wenn *bh* ein gültiger Binomial-Heap ist:

```
1 def isBinomHeap(bh):
2   n=len(bh)-1
3   return all(isBinomTreeK(n-i,bt) for i,bt in enumerate(bh) if bt)
```

Aufgabe 4.7

Implementieren Sie eine Funktion *insertBinomialheap*(*bh*,*x*) die als Ergebnis einen Binomial-Heap zurückliefert, der durch Einfügen von x in *bh* entsteht.

Lösung

Folgende Pythonfunktion implementiert das Einfügen eines Elements x in einen Binomial-Heap *bh*:

```
1 def insertBinomialheap(bh,x):
2   return merge(bh,[(x,[])])
```

Folgendermaßen kann man etwa einen Binomial-Heap mit 100 Elementen erzeugen:

```
from random import randint
bh=[(randint(1,1000),[])]
for _ in range(22):
  bh = insertBinomialheap(bh,randint(1,1000))
```

Aufgabe 4.8

...

Lösung

Folgende Pythonfunktion implementiert das Einfügen eines Elements x in einen Binomial-Heap *bh*:

```
def insertBinomialheap(bh,x):
  return merge(bh,[(x,[])])
```

Folgendermaßen kann man etwa einen Binomial-Heap mit 100 Elementen erzeugen:

```
from random import randint
bh=[(randint(1,1000),[])]
for _ in range(22):
  bh = insertBinomialheap(bh,randint(1,1000))
```

Aufgabe 4.9

Erklären Sie, warum der Knoten mit minimalem Schlüsselwert sich immer an der Wurzel eines Fibonacci-Baums befinden muss.

Lösung

Warum muss sich der Knoten mit minimalem Schlüsselwert immer an der Wurzel eines Fibonacci-Baums befinden? – Wäre dies nicht der Fall, so müsste ein Elternknoten einen größeren Wert enthalten als dieser minimale Knoten. Das geht aufgrund der Min-Heap-Bedingung nicht.

Aufgabe 4.10

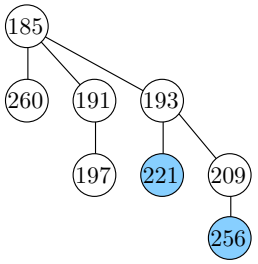
Vervollständigen Sie den oben gezeigten Wert so, dass er den in Abbildung **??** gezeigten Fibonacci-Heap vollständig repräsentiert.

Lösung

{*treesFH*: [{*rootFT*: 59, *subtreesFT*: [{*rootFT*: 88, *subtreesFT*: [] , *marked*: *False*}], *marked*: *False*},
{*rootFT*: 30, *subtreesFT*: [{*rootFT*: 80, *subtreesFT*: [], *marked*: *False*}, {*rootFT*: 85, *sub*
{*rootFT*: 40, *subtreesFT*: [], *marked*: *False*},
{*rootFT*: 65, *subtreesFT*: [], *marked*: *False*}],
minFH: 1}

Aufgabe 4.11

(a) Implementieren Sie eine Funktion *FT2str*(*ft*), die aus einem Fibonacci-Baum eine gut lesbare Stringform produziert. Schreiben Sie die Funktion so, dass etwa aus dem rechts dargestellten Fibonacci-Baum der folgende String produziert wird:




```
'185-(260 ; 191-197 ; 193-(#221 ; 209-#256))'
```

Die Liste der Teilbäume soll also immer in runden Klammern eingeschlossen sein; die einzelnen Teilbäume sollen durch `';'` getrennt sein; markierten Knoten soll ein `'#'` vorangestellt werden.

(b) Implementieren Sie eine Funktion *FH2str*(*fh*), die aus einem Fibonacci-Heap eine gut lesbare Stringform produziert; verwenden Sie hierzu die in der letzten Teilaufgabe beschriebene Funktion *FT2str*.

Lösung

(a) Eine Funktion *FT2str*(*ft*), die aus einem Fibonacci-Baum eine gut lesbare Stringform wie in der Aufgabenstellung beschrieben produziert:

```
1 def FT2str(ft):
2   if ft[markedFT]:
3     s = "#" + str(ft[rootFT])
4   else:
5     s = str(ft[rootFT])
6   ss = map(FT2str,ft[subtreesFT])
7   if len(ss)>1: s += '-(' + ' ; '.join(ss) + ')'
8   elif len(ss)==1: s += '-' + ss[0]
9   return s
```

(b) Eine Funktion *FH2str*(*fh*), die einen Fibonacci-Heap in eine gut lesbare Stringform umwandelt.

```
1 def FH2str(fh):
2   return str(map(FT2str,fh[treesFH]))
```

Aufgabe 4.12

Schreiben Sie eine Funktion *FH2List*, die die in einem Fibonacci-Heap enthaltenen Elemente als Liste zurückliefert.

Lösung

Mittels der folgenden Funktionen kann man einen Fibonacci-Heap in eine Liste umwandeln:

```
1  def FH2List(fh):
2    xs = []
3    for ft in fh[treesFH]:
4      xs += FT2List(ft)
5    return xs
6
7  def FT2List(ft):
8    xs = [ft[rootFT]]
9    for t in ft[subtreesFT]:
10     xs += FT2List(t)
11   return xs
```

Aufgabe 4.13

Die in Listing **??** gezeigte Implementierung stellt eine nicht-destruktive Realisierung der Verschmelzungs-Operation dar. Implementieren Sie eine destruktive Version *mergeFHD*(*fh1*,*fh2*), die keinen „neuen“ Fibonacci-Heap als Rückgabewert erzeugt, sondern nichts zurückliefert und stattdessen den Parameter *fh1* (destruktiv) so verändert, dass dieser nach Ausführung von *mergeFHD* den Ergebnis-Heap enthält.

Lösung

Implementierung einer desktruktiven Version *mergeFHD*(*fh1*,*fh2*), die keinen „neuen“ Fibonacci-Heap als Rückgabewert erzeugt, sondern nichs zurückliefert und stattdessen den Parameter *fh1* (destruktiv) so verändert, dass dieser nach Ausführung von *mergeFHD* den Ergebnis-Heap enthält:

```
1 def mergeFHD(fh1,fh2):
2   f1[treesFH].append(fh2[treesFH])
3   if getMinFH(fh1) > getMinFH(fh2):
4     fh1[minFH] = len(fh1[treesFH]) -1 +fh2[minFH]
```

Aufgabe 4.14

Implementieren Sie die Funktion *makeFT*, die in Zeile 2 in Listing **??** benötigt wird.

Lösung

Folgende Funktion liefert einen einfachen Fibonacci-Baum zurück, der lediglich ein Element x enthält.

```
def makeFT(x):
  return { rootFT : x , subtreesFT : [] }
```

Aufgabe 4.15

Die *insert*-Funktion aus Listing **??** ist destruktiv, d. h. sie verändert ihr Argument *fh* und liefert keinen Wert zurück. Implementieren Sie eine nicht-destruktive Variante dieser *insert*-Funktion, die ihr Argument *fh* nicht verändert und stattdessen einen neuen Fibonacci-Heap zurückliefert, in den das Element x eingefügt wurde.

Lösung

Folgende Funktion implementiert die nicht-destruktive Einfügeoperation:

```
def insertND(x,fh):
  ft = makeFT(x)
  if getMinFH(fh) > x: # min-Pointer anpassen
    i = len(fh[treesFH])
  else:
    i = fh[minFH]
  return {treesFH : fh[treesFH] + [ft] , minFH : i }
```

Aufgabe 4.16

Implementieren Sie die in Zeile 9 in Listing **??** benötigte Funktion *mergeFT*, die zwei Fibonacci-Bäume *ft1* und *ft2* so verschmilzt, dass die Heap-Bedingung erhalten bleibt.

Lösung

Folgendes Listing zeigt die Implementierung der Funktion *mergeFT*, die in Listing **??** (Zeile 9) benötigt wird.

```
1 def mergeFT(ft1,ft2):
2   if ft1[rootFT] < ft2[rootFT]:
3     return { rootFT : ft1[rootFT] , subtreesFT : ft1[subtreesFT] + [ft2] }
4   else:
5     return { rootFT : ft2[rootFT] , subtreesFT : ft2[subtreesFT] + [ft1] }
```

Aufgabe 4.17

Schreiben Sie eine Python-Funktion *isConsistent*(*fh*), die überprüft, ob die Vorwärts- und Rückwärtsverzeigerung in allen Bäumen eines Fibonacci-Heaps *fh* konsistent ist.

Lösung

Die Funktion *isConsistent*(*fh*), die überprüft, ob Vorwärts- und Rückwärtsverzeigerung konsistent sind, kann wie folgt implementiert werden:

```
1 def isConsistent(fh):
2   return all(isConsistentFT(t) for t in fh[treesFH])
3
4 def isConsistentFT(ft):
5   return all(isConsistentFT(t) for t in ft[subtreesFT]) and \
6          all( t[parentFT]==ft for t in ft[subtreesFT])
```

Aufgabe 4.18

Erstellen Sie eine Funktion *allPaths*(*fh*), die die Liste aller gültigen Pfade eines Fibonacci-Heaps erzeugt – und zwar so, dass jeder dieser Pfade als möglicher zweiter Parameter der in Listing **??** gezeigten Funktion *decKey* dienen könnte.

Lösung

Der Schlüsselwert eines zufällig ausgewählten Elements eines Fibonacci-Heaps kann mit Hilfe von *allPaths* folgendermaßen erniedrigt werden:

```
>>> from random import choice
>>> delta = ...
>>> fh = ...
>>> decKey(fh,choice( allPaths(fh), delta)
```

Aufgabe 4.21

Schreiben Sie eine Funktion *ph2str*, die einen Pairing-Heap als Argument übergeben bekommt und eine gut lesbare String-Repräsentation dieses Pairing-Heaps zurückliefert. Die String-Repräsentation des Pairing-Heaps aus Abbildung **??** sollte hierbei beispielsweise folgende Form haben:

```
'26-[48-49-[99,95],74,50-61,73,31-[39,69]]'
```

Die Teilbaumlisten sollten also – vorausgesetzt sie bestehen aus mehr als einem Baum – in eckige Klammern eingeschlossen werden; das Wurzelelement sollte mit einem `'-'` von seiner Teilbaumliste getrennt sein.

Lösung

Folgende Funktion *ph2str*, die einen Pairing-Heap als Argument übergeben bekommt, liefert die geforderte String-Repräsentation dieses Pairing-Heaps zurück.

```
1 def ph2str(ph):
2   if ph:
3    if len(ph[subtreesPH])==0: subs=''
4    elif len(ph[subtreesPH])==1: subs='-' + ph2str(ph[subtreesPH][0])
5    else: subs = '-[' + ','.join(map(ph2str,ph[subtreesPH])) + ']'
6    return str(ph[rootPH]) + subs
7   else: return ''
```

Aufgabe 5.1

Erweitern Sie die Klasse *Graph* um die Methode *Graph.w(i,j)*, die das Gewicht der Kante (i, j) zurückliefert (bzw. *None*, falls die Kante kein Gewicht besitzt).

Lösung

Die Implementierung der drei geforderten Methoden, nämlich $V()$ um sich die Liste der Knoten zurückliefern zu lassen, $E()$ um sich die Liste aller Kanten zurückliefern zu lassen und $w(i, j)$ um sich das Gewicht der Kante (`i`, `j`) zurückliefern zu lassen.

```
def w(self, i, j):
  return self.vertices[i][j]

def V(self):
  return [i for i in range(0, self.numNodes+1)]
```

```
def E(self):
  return [(i,j) for i in self.V() for j in self.vertices[i].keys()]
```

Aufgabe 5.2

Erweitern Sie die Klasse *Graph* um die folgenden Methoden:

(a) Eine Methode *Graph.isPath*(*vs*), die eine Knotenliste *vs* übergeben bekommt und prüft, ob es sich hierbei um einen Pfad handelt.

(b) Eine Methode *Graph.pathVal*(*vs*), die eine Knotenliste *vs* übergeben bekommt. Handelt es sich dabei um einen gültigen Pfad, so wird der „Wert“ dieses Pfades (d. h. die Summe der Gewichte der Kanten des Pfades) zurückgeliefert. Andernfalls soll der Wert ∞ (in Python: *float*('inf')) zurückgeliefert werden. Verwenden Sie hierbei das folgende „Gerüst“ und fügen Sie an der mit „ ... “ markierten Stelle die passende Listenkomprehension ein.

```
def pathVal(self, xs):
  if len(xs)<2: return 0
  return sum([...])
```

Lösung

(a) Implementierung der Methode *Graph.isPath*(*vs*):

```
class Graph(object):
  ...
  def isPath(self,vs):
    if len(vs)<2: return True
    return all([self.isEdge(v[i],v[i+1]) for i in range(len(vs)-1)])
```

Wobei die Python-Funktion *all* die Und-Verknüpfung einer Sequenz boolscher Werte zurückliefert.

(b) Implementierung der Methode *Graph.pathVal*(*vs*) geht ganz ähnlich, außer dass hier die Werte der Kanten des Pfades aufsummiert werden.

```
def pathVal(self, xs):
  if len(xs)<2: return 0
  return sum([self.w(xs[i],xs[i+1]) for i in range(len(xs)-1)])
```

Aufgabe 5.4

Implementieren Sie eine Klasse *Queue*, die die Operationen *enqueue*(*x*), *dequeue*() und *isEmpty* unterstützt.

Lösung

Eine Queue kann man in Python folgendermaßen einfach implementieren:

```
1  class Queue(object):
2    def __init__(self):
3      self.q = []
4    def enqueue(self,x):
5      self.q.append(x)
6    def dequeue(self):
7      retVal = self.q[0]
8      del(self.q[0])
9      return retVal
10   def isEmpty(self):
11     return self.q==[]
```

Aufgabe 5.7

Es gibt eine entscheidende Ineffizienz in der in Listing **??** vorgestellten Implementierung der Tiefensuche: Obwohl in jedem Schleifendurchlauf der **while**-Schleife nur *ein einziger* noch nicht besuchter Nachbar von v zur weiteren Bearbeitung benötigt wird, wird in der Listenkomprehension in Zeile 8 immer die gesamte Menge der noch nicht besuchten Nachbarn berechnet.

Verbessern sie die Implementierung der Tiefensuche, indem sie diese Ineffizienz entfernen.

Lösung

Man ersetze die Zeile 8 aus Listing **??** durch folgenden Code:

```
8    for u in graph.G(v):
9      if pred[u])==None and u≠s: break
```

So erzeugt man nicht die Menge aller unbesuchten Nachbarn, sondern sucht nur solange bis man einen gefunden hat.

Aufgabe 5.9

Statt explizit einen Stack zu verwenden, lässt sich die Tiefensuche elegant rekursiv implementieren. Implementieren Sie eine rekursive Variante *dfsRek*, des in Listing **??** gezeigten Algorithmus *dfs*.

Lösung

Listing 8 zeigt eine rekursive Implementierung der Tiefensuche.

```
1  def dfs(s, graph):
2    pred = []
3    n = graph.numNodes
4    for i in range(0,n): pred.append(None)
5    def visit(i, pred):
6      for j in graph.G(i):
7        if pred[j] == None and j≠s:
8          pred[j]=i
9           visit(j, pred)
10    visit(s, pred)
11   return pred
```

Listing 8: Rekursive Implementierung der Tiefensuche

Aufgabe 5.11

Beim Anziehen von Kleidungsstücken müssen manche Teile unbedingt vor anderen angezogen werden. Die folgenden Beziehungen sind vorgegeben:

- Das *Unterhemd* vor dem *Pullover*
- Die *Unterhose* vor der *Hose*
- Den *Pullover* vor dem *Mantel*
- Die *Hose* vor dem *Mantel*
- Die *Hose* vor den *Schuhen*
- Die *Socken* vor den *Schuhen*

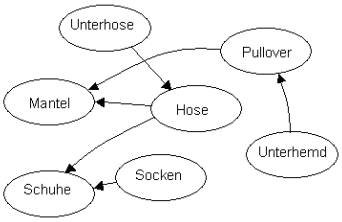
(a) Modellieren Sie diese Abhängigkeiten als Graphen.

(b) Nummerieren Sie die Knoten so, dass sich die daraus ergebende Rangordnung der Knoten eine topologische Sortierung darstellt – gibt hier mehrere Lösungen.

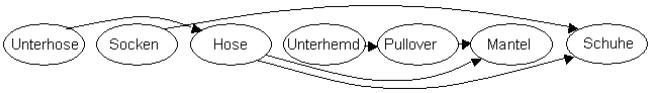
(c) Bestimmen Sie diejenige topologische Sortierung, die sich durch Ausführung von dem in Listing **??** gezeigten Algorithmus ergibt.

Lösung

(a) Folgender Graph ist eine Modellierung der Beziehungen:



(b) So kann man eine topologische Sortierung darstellen. Der Knoten ganz links hat die kleinste Nummer, der Knoten ganz rechts die größte. Die Pfeile verlaufen nur von links nach rechts.



(c) ...

Aufgabe 5.12

Die topologische Sortierung erwartet als Eingabe einen Knoten, der keinen Vorgänger besitzt. Implementieren Sie eine Funktion *startNodes*(*graph*), die alle Knoten des Graphen *graph* zurückliefert, die keinen Vorgängerknoten besitzen.

Lösung

Ein Lösung sieht so aus, dass wir für jeden Knoten $i \in V$ prüfen, ob es einen (von i verschiedenen) Knoten j gibt von dem aus eine Kante nach i geht. Dies lässt sich kompakt in einer Listenkomprehension ausdrücken:

```
1 def startNodes(graph):
2   return [i for i in graph.V()
3             if  [j for j in graph.V() if graph.isEdge(j,i) and i≠j] == []]
```

Diese Lösung hat eine Laufzeit von $O(|V|^2)$.

Für weniger dichte Graphen (h. h. für Graphen mit $|E| << |V|^2$) gibt es jedoch eine schnellere Lösung:

```
1 def startNodes2(graph):
2   flags = [0] + [1] *graph.numNodes
3   for (i,j) in graph.E():
4     flags[j]=0
5   return flags.index(1)
```

Diese Lösung vermeidet, es alle Kombinationen von Knotenpaaren aus $V \times V$ zu betrachten, sondern läuft lediglich über die Menge der Kanten und hat daher nur eine Laufzeit von $O(|E|)$. Das Array *flags* enthält zu Beginn nur Einsen; immer dann, wenn wir eine Kante gefunden haben, die bei Knoten j endet, tragen wir in *flags* $[j]$ den Wert 0 ein (was soviel heißt wie: der Knoten j gehört nicht zu Menge der Knoten, die keinen Vorgänger besitzen.

Aufgabe 5.14

In jedem der $|V|$ vielen Durchläufe des Dijkstra-Algorithmus muss der Knoten mit minimalem Abstandswert l bestimmt werden. Dies geschieht in Algorithmus **??** mittels der *min*-Anweisung in Zeile 9.

(a) Welche Laufzeit hat diese *min*-Anweisung?

(b) Statt das Minimum aus einer Liste zu bestimmen ist es i. A. effizienter ein Heap-Datenstruktur zu verwenden und mittels *minExtract* das Minimum zu extrahieren. Welche Laufzeit hätte das Finden des Knotens mit minimalem Abstandswert, falls statt einer einfachen Liste eine Heap-Datenstruktur verwendet wird?

(c) Geben sie eine Python-Implementierung des Dijkstra-Algorithmus an, zum Finden des minimalen Abstandswertes Heaps verwendet.

Lösung

(a) Die *min*-Anweisung hat die Laufzeit $O(|V|)$, denn es müssen i.A. $|V|$ viele l-Werte verglichen werden, um deren Minimum zu finden.

(b) Falls eine Heap-Datenstruktur verwendet wird, hat das Finden (und gleichzeitiges Extrahieren) des Miniumums eine Laufzeit von $O(\log |V|)$.

(c) Eine Optimierung kann folgendermaßen aussehen:

(d) Der gesamte Algorithmus hätte mit dieser Optimierung folgende Laufzeit: Zunächst wird der Heap mit den Knoten aus V initialisiert (Laufzeit $O(|V|)$); in den $O(|V|)$ vielen Durchläufen durch den Dijkstra-Algorithmus wird jedesmal das Minimum dieses Heaps entfernt – insgeamt ergibt dies eine Laufzeit von $O(|V| \log |V|)$. Jedesmal, wenn ein l-Wert aktualisiert wird muss dieser Wert in der Heap-Datenstruktur nach oben wandern (Laufzeit $O(\log |V|)$); dies geschieht $O(|E|)$ mal, was eine Laufzeit von $O(|E| \log |V|)$ ergibt. Insgesamt erhalten wir also eine Laufzeit von $O((|E| + |V|) \log |V|)$.
Ohne diese Optimierung hätte der Dijkstra-Algorithmus eine Laufzeit von $O(|V|^2)$, was bei relativ dünnen Graphen deutlich langsamer ist.

(e)

```
1 def dijkstraHeap(u,graph):
2     n = graph.numNodes
3     unendlich = float('inf')
```

```
    W = graph.V()
    l = [ (0 if v==u else unendlich, v) for v in graph.V() ]
    l_erg = []
    heapify(l)
    F = [] ; k = {}
    for i in range(0,n):
        lv,v = heappop(l)
        l_erg.append((v,lv))
        W.remove(v)
        if v≠u: F.append(k[v])
        for j in range(len(l)):
            lv1, v1 = l[j]
            if v1 in graph.G(v):
                if lv + graph.w(v,v1) < lv1:
                    l[j] = (lv + graph.w(v,v1), v1)
                    k[v1] = (v,v1)
        heapify(l)
    return l_erg,F
```

Aufgabe 5.18

Eine verbesserte Implementierung des Kruskal-Algorithmus würde es vermeiden die gesamte Kantenmenge zu sortieren, sondern stattdessen einen Heap verwenden, um in jedem Durchlauf effizient die Kante mit dem minimalen Gewicht auszuwählen.

Passen Sie die Implementierung des in Listing **??** gezeigten Skripts entsprechend an.

Lösung

Die Implementierung es Kruskal-Algorithmus unter Verwendung von Heaps:

```
from heapq import buildHeap,heappop

def kruskal(graph):
    allEdges = [(graph.w(i,j), i, j) for i,j in graph.E()]
    buildHeap(allEdges)
    spannTree = []
    while len(spannTree) ≠len(graph.V()) -1:
        (w,i,j) = heappop(allEdges)
        if not buildsCircle(spannTree,(i,j)):
            spannTree.append((i,j))
    return spannTree
```

Aufgabe 5.21

(a) Kann man den minimalen Spannbaum auch finden, indem man genau umgekehrt wie der Kruskal-Algorithmus vorgeht, d. h. man beginne mit allen im Graphen enthaltenen Kanten und entfernt Kanten mit dem momentan höchsten Gewicht – aber nur dann, wenn man dadurch den Graphen nicht auseinanderbricht?

(b) Geben Sie eine Implementierung des „umgekehrten“ Kruskal-Algorithmus an.

Lösung

(a) Wir können zwei Fälle unterscheiden:

1. Wird eine Kante e entfernt und wird der Graph durch das Entfernen dieser Kante nicht auseinander gerissem, so muss es vor Enfernen dieser Kante einen Kreis C gegeben haben; die Kante e ist – da dies die erste zu entfernende Kante dieses Kreises ist – die Kante mit maximalem Gewicht dieses Kreises C. Nach der Kreiseigentschaft kann e also nicht zum minimalen Spannbaum gehören und somit wird diese Kante „zu Recht“ entfernt.
2. Wird der Graph durch Entfernen der Kante e mit (momentan) maximalem Gewicht auseinander gerissen – in zwei Teile A und $V \setminus A$ – so ist e die Kante mit minimalem Gewicht in $e(A)$ (denn alle anderen Kanten in $e(A)$ wurden ja schon vorher entfernt, haben also größeres Gewicht. Somit gehört e zum minimalen Schnitt und wird „zu Recht“ nicht entfern.

Somit ist gezeigt: Auch der „umgedrehte“ Kruskal-Algorithmus führt zum minimalen Spannbaum.

(c) Implementierung des „umgedrehten“ Kruskal-Algorithmus:

```
1 def kruskalBackward(graph):
2   allEdges = [(graph.w(i,j), i,j) for i,j in graph.E_undir()]
3   allEdges.sort() ;
4   spannTree = allEdges
5   while len(spannTree) > len(graph.V()) -1 and allEdges≠[]:
6     (w,i,j) = allEdges.pop()
7     if not breaks(spannTree,(i,j)):
8       spannTree.append((i,j))
9   return spannTree
```

Es fehlt nur die Implementierung von *breaks* - die aber leicht über eine Tiefen- oder Breitensuche realisiert werden kann - man startet einfach bei einem beliebigen Knoten und testet, ob nach der Tiefen-/Breitensuche alle Einträge in *pred* einen Wert ungleich *None* haben.

Aufgabe 5.22

Implementieren Sie für die in Listing **??** gezeigte Klasse *UF* die *str*-Funktion, die ein Objekt der Klasse in einen String umwandelt. Die Ausgabe sollte gemäß folgendem Beispiel erfolgen:

```
>>> uf = UF(10)
>>> uf.union(1,2) ; uf.union(1,3) ; uf.union(5,6) ; uf.union(8,9)
>>> str(uf)
>>> '{1, 2, 3} {4} {5, 6} {7} {8, 9} '
```

Lösung

Folgendermaßen kann eine sinnvolle *str*-Funktion für den die *UF*-Klasse entworfen werden:

```
1  def __str__(self):
2      retStr=""
3      def take(i, lst):
4          if lst==[] or lst[0][0]≠i: return []
5          else: return [lst[0]] + take(i, lst[1:])
6      allNodes = [(self.find(i), i) for i in range(1, self.n)]
7      allNodes.sort()
8      while allNodes ≠ []:
9          retStr=retStr+'{'
10         i = allNodes[0][0]
11         nodeSet = map(lambda (x,y): y, take(i,allNodes))
12         retStr=retStr+str(nodeSet)[1:-1]
13         allNodes=allNodes[len(nodeSet):]
14         retStr=retStr+'} '
15     return retStr
```

Die Variable *allNodes* enthält alle verwalteten Elemente der Union-Find-Datenstruktur, zusammen mit der einheitlichen Repräsentation derjenigen Menge der sie angehören (abrufbar über die *find*-Funktion). Sortiert man diese Liste, so stehen die Elemente sortiert nach Zugehörigkeit ihrer jeweiligen Mengen. Nun muss nur noch über die Hilfsfunktion *take* diejenigen Elemente gemeinsam aus *allNodes* entfernt werden, die denselben *find*-Wert besitzen. nun immer die nächsten Elemente mit gemein

Aufgabe 5.23

Verbessern Sie die in Abbildung **??** gezeigte Implementierung dadurch, dass Sie auf die Balancierung der in der Union-Find-Datenstruktur verwalteten Bäume achten. Der Baum *find*(x) sollte also nur dann als Kind unter die Wurzel des Baums *find*(y) gehängt

werden, wenn die Höhe von *find*(x) kleiner ist als die Höhe von *find*(y); andernfalls sollte *find*(y) unter die Wurzel von *find*(x) gehängt werden.

Lösung

Die Implementierung der Union-Find-Datenstruktur mit Berücksichtigung der Balancierung der Bäume:

```
1  class UF1(object):
2    # Weight-Balanced
3    def __init__(self,n):
4      self.n = n
5      self.parent = [0]*n
6    def find(self,x):
7      while self.parent[x] > 0: x = self.parent[x]
8      return x
9    def union(self,x,y):
10     # nur dem den Roots!
11     if (-self.parent[x])>=(-self.parent[y]):
12       if self.parent[x]==self.parent[y]:
13         self.parent[x] += -1
14       self.parent[y] = x;
15     else:
16       self.parent[x] = y
```

Die Implementierungen von *find* und *__init__* bleiben unverändert. Bei der Implementierung von *union* wird nun darauf geachtet, dass immer kleinere (oder gleich große) Bäume unter Wurzel eines grösßeren (oder gleich großen) Baumes gehängt werden. Die Wurzel x eines Baumes enthält in $parent[x]$ immer die Höhe gespeichert als negative Zahl (um den entsprechenden Knoten auch als Wurzelknoten erkennen zu können). Immer dann, wenn zwei gleich große Bäume zusammengehängt werden, vergrößert sich die Höhe des Baumes um 1 (die entsprechende Anpassung wird in Zeile 13 vorgenommen); in allen anderen Fällen verändert sich die Höhe offensichtlich nicht.

Durch die im folgenden Listing gezeigte kleine Anpassung der *__str__*-Funktion kann man die Höhen der in der Union-Find-Datenstruktur erzeugten Bäume mit anzeigen lassen:

```
1  class UF1(object):
2    ...
3    def __str__(self):
4      retStr=""
5      def take(i, lst):
6        if lst==[] or lst[0][0]!=i: return []
7        else: return [lst[0]] + take(i, lst[1:])
```

```
8       allNodes = [(self.find(i), i) for i in range(1, self.n)]
9       allNodes.sort()
10      while allNodes ≠ []:
11          retStr=retStr+ '{'
12          i = allNodes[0][0]
13          nodeSet = map(lambda (x,y): y, take(i,allNodes))
14          height = min(map(lambda x: self.parent[x], nodeSet))
15          retStr=retStr+ str(nodeSet)[1:-1]
16          allNodes=allNodes[len(nodeSet):]
17          retStr=retStr+ '}' + str(-height) + '  '
18      return retStr
```

In Zeile 14 wird die Höhe des Baumes, der die aktuell bearbeitete Menge repräsentiert, extrahiert (die einfach der minimale *parent*-Eintrag in den Knoten der jeweiligen Menge ist, h. h. der *parent*-Eintrag der Wurzel).

Aufgabe 5.27

Implementieren Sie die in Zeile 17 in Listing **??** benötigte Funktion *findPath*, die nach einem gültigen Pfad von s nach t im Restnetzwerk sucht.

Hinweis: Um das in Abbildung **??** erwähnte Problem zu vermeiden, muss eine Breitensuche verwendet werden – erklären Sie warum!

Lösung

Die Funktion *findPath* kann analog der Breitensuche folgendermaßen implementiert werden:

```
1  def findPath(s, t, graph):
2    q = Queue() ; d = [] ; pred = []
3    for i in range(0,graph.numNodes+1):
4      d.append( -1 if i≠s else 0); pred.append(None)
5    v = s
6    while v ≠None:
7      unvisited = [ u for u in graph.G(v) if d[u]==-1 ]
8      if unvisited ≠ []:
9        for u in unvisited:
10         d[u] = d[v] +1
11         pred[u] = v
12         q.enqueue(u)
13         if u==t:
14           path = []
15           while t≠s:
16             path.append((pred[t], t))
```

```
17              t=pred[t]
18            path.reverse()
19            return path
20      elif not q.isEmpty():
21        v = q.dequeue()
22      else:
23        v = None
24    return []
```

Die Implementierung ist praktisch die Gleiche wie die von *bfs* bis auf den folgenden kleinen Unterschied: sobald der Knoten t „gefunden“ wurde bricht die Suche ab. Der Rückgabewert *path*, also der Pfad von s nach t wird einfach dadruch erzeugt, indem das durch die Breitensuche erzeugte *pred*-Feld vom Knoten t rückwärts bis zu Knoten s gelaufen wird. Dies geschieht in den Zeilen 15 bis 19.

Aufgabe 5.29

(a) Definieren Sie eine Python-Funktion *cut*(*C*,*graph*), die eine den Schnitt definierende Knotenmenge C und einen Graphen *graph* übergeben bekommt und eine Liste aller Kanten zurückliefert die sich im Schnitt befinden. Versuchen Sie eine Implementierung als „Einzeiler“, also in der Form

```
def cut(C,graph):
  return ...
```

(b) Definieren Sie eine Python-Funktion *cutVal*(*C*,*graph*), die den Wert des Schnittes zurückliefert, der durch die Knotenmenge C definiert ist. Versuchen Sie wiederum eine Implementierung als Einzeiler.

Lösung

(a) Die Implementierung von *cut*:

```
def cut(C,graph):
  return [(i,j) for i,j in graph.E_undir() if i in C and j not in C]
```

(b) Die Implementierung von *cutVal*:

```
def cutVal(C,graph):
  return sum( [graph.w(i,j) for i,j in cut(C,graph)] )
```

Aufgabe 5.30

Zeigen Sie die eben aufgestellte Behauptung, die besagt dass – für einen gegebenen Fluss f – der Fluss jedes beliebigen Schnittes S immer denselben Wert hat.

Lösung

Dies kann man, wie vor der Aufgabenstellung erwähnt, über die Induktion über die Anzahl der Knoten im s-t-Schnitt A zeigen:

Induktionsanfang: $|A| = 1$, also $A = \{s\}$. Hier gilt, dass $f(A) = f$ (laut Definition des Flusses).

Induktionsschritt: $|A| = n + 1$. Entfernt man einen beliebigen Knoten v aus A, so ... verändert sich der Schnitt $f(A)$ dadurch folgendermaßen:

$$f(A \setminus \{v\}) = f(A) - \sum_{u \notin A} f(v,u) + \sum_{u \in A} f(u,v) = f(A) - \sum_{u \in V} f(u,v) = f(A) - 0 = f(A)$$

Durch Wegnehmen einer Kante verändert sich also der Fluss des Schnittes nicht; der Induktionsschritt wäre also hiermit gezeigt.

Aufgabe 6.1

Geben Sie den Wert der folgenden Ausdrücke an:

(a) $\{\varepsilon\}^*$ (b) $|\{\ w \in \{a,b,c\}^* \mid |w| = 2\ \}|$ (c) $|\{0,1\}^*|$

Lösung

(a) $\{\varepsilon\}$

(b) 6

(c) ∞

Aufgabe 6.3

Erweitern Sie die Grammatik so, dass alle (möglicherweise geschachtelte) Ziffer-Tupellisten (in Python-Notation) erzeugt werden. Folgende Wörter sollten beispielsweise in der durch die Grammatik erzeugten Sprache enthalten sein:

`([1,(1,2)],(2,),[2],[],())` `([1],)` `[1,2]` `(1,[2])`

Beachten Sie, dass ein-elementige Tupel mit einem Komma am Ende notiert werden.

Lösung

Die folgende Grammatik beschreibt alle korrekten Tupel-Listen-Ausdrücke (die Ziffern enthalten):

$$
\begin{array}{rcl}
\mathit{TListe} & \rightarrow & \texttt{[}\ \mathit{elemente}\ \texttt{]} \mid \texttt{(}\ \mathit{elemente}\ \texttt{)} \\
& \mid & \texttt{[}\ \mathit{element}\ \texttt{]} \mid \texttt{(}\ \mathit{element}\ \texttt{,}\ \texttt{)} \\
& \mid & \texttt{[]} \mid \texttt{()} \\
\mathit{element} & \rightarrow & \mathit{element}\ \texttt{,}\ \mathit{element} \mid \mathit{element}\ \texttt{,}\ \mathit{elemente} \\
\mathit{element} & \rightarrow & \mathit{TListe} \mid \mathit{ziffer} \\
\mathit{ziffer} & \rightarrow & \texttt{0} \mid \ldots \mid \texttt{9}
\end{array}
$$

Aufgabe 6.5

Schreiben Sie für die Klasse *Grammatik* die Methode *__repr__*, um eine angemessene String-Repräsentation einer Grammatik zu definieren. Orientieren Sie sich an folgender Ausgabe:

```
>>> print G
D --> E
E --> E + T
E --> T
T --> T * F
T --> F
F --> ( E )
F --> id
```

Lösung

Folgende Implementierung von *__repr__* liefert die gewünschte String-Repräsentation einer Grammatik:

```
1 class Grammatik(object):
2     ...
3   def __repr__(self):
4       erg = ''
5       for p in self.P:
6           erg += p[0] +' --> ' +' '.join(p[1]) +'\n'
7       return erg[:-1]
```

Die Variable p durchläuft alle Produktionen in *self.P*. In der String-Variablen *erg* werden die String-Repräsentationen der einzelnen Variablen akkumuliert. Hierbei ist $p[0]$

immer jeweils das Nichtterminal der linken Seite einer Produktion; $p[1]$ ist immer jeweils die Liste der Symbole der rechten Seite einer Produktion; diese werden mittels `' '.join(p[1])` mit einem Leerzeichen getrennt zu einem String zusammengefügt.

Aufgabe 6.7

Wo und wie genau wird der Fall 1. in dem in Abbildung **??** dargestellten Algorithmus in der in Listing **??** Implementierung abgedeckt.

Lösung

Der Fall 1 ist durch die Anweisung

```
if all([Y in self.V and '' in self.first[Y] for Y in alpha]):
    self.first[X].add('')
```

voll abgedeckt: Ist die rechte Seite der entsprechenden Produktion leer (also gleich ε), so ist das Argument von *all* die leere Liste; da *all*([]) sich zu *True* auswertet, wird in diesem Fall `''` (entspricht ε) zu *first* [*X*] hinzugefügt.

Aufgabe 6.8

Erstellen Sie eine Methode *firstSatzform* der Klasse *Grammatik*. Hierbei soll *firstSatzform*(α) die FIRST-Menge der Satzform α zurückliefern.

Lösung

```
1  class Grammatik(object):
2    ...
3    def firstSatzform(self,w):
4      if not w: return set([''])
5      if not all(c in self.V+self.T for c in w):
6        print 'Error!!!'
7        return
8      erg = set()
9      for Y in w:
10       if Y in self.T:
11         erg.add(Y)
12       if Y in self.V:
13         erg = erg.union(self.first[Y])
14       if Y in self.T or '' not in self.first[Y]: break
15     if all(Y in self.V and '' in self.first[Y] for Y in w): erg = erg.add('')
16     return erg
```

Aufgabe 6.10

Gegeben sei die Grammatik $G = (Z, \{a, b, c\}, \{Z, S, A, B\}, P\}$, wobei P aus den folgenden Produktionen besteht:

$$
\begin{aligned}
Z &\rightarrow S \mid \varepsilon \\
S &\rightarrow BASc \mid aSa \\
A &\rightarrow bAb \\
B &\rightarrow cBc \mid \varepsilon
\end{aligned}
$$

Berechnen Sie die FOLLOW-Mengen aller Nichtterminale.

Lösung

```
>>> G3.follow
{'A': set(['a', 'c', 'b', '$']),
 'S': set(['a', 'c', '$']),
 'Z': set(['$']),
 'B': set(['c', 'b'])}
```

Aufgabe 6.12

Gegeben sei die folgende linksrekursive Grammatik

$$
\begin{aligned}
ausdr &\rightarrow ausdr \texttt{ + } term \\
ausdr &\rightarrow ausdr \texttt{ - } term \\
ausdr &\rightarrow term \\
term &\rightarrow \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{9}
\end{aligned}
$$

(a) Eliminieren Sie die Linksrekursion aus dieser Grammatik.

(b) Implementieren Sie einen Recursive-Descent-Parser, der die durch diese Grammatik beschriebene Sprache erkennt.

Lösung

$$
\begin{aligned}
ausdr &\rightarrow term\ rest \\
rest &\rightarrow \texttt{+}\ term\ rest \mid \texttt{-}\ term\ rest \\
rest &\rightarrow \varepsilon \\
term &\rightarrow \texttt{0} \mid \ldots \mid \texttt{9}
\end{aligned}
$$

Aufgabe 6.13

Implementieren Sie für die Klasse *Grammatik* eine Methode *printElement*(i,j), das das durch (i,j) repräsentierte LR(0)-Element in gut lesbarer Form auf dem Bildschirm ausgibt, wie etwa in folgender Beispielanwendung:

```
>>> G.printElement(5,1)
'F -> ( . E )'
```

Lösung

Folgendermaßen kann ein als Tupel repräsentiertes LR(0)-Element in gut lesbarer Form als String ausgegeben werden.

```
1 def flat(xs): return reduce(lambda x,y: x+' '+y, xs, '')
2 class Grammatik(object):
3     ...
4   def printElement(self,i,j):
5       p = self.P[i]
6       return p[0] +' -> ' +flat(p[1][:j]) +' .' +flat(p[1][j:])
```

Aufgabe 6.14

Durchläuft der in Listing **??** gezeigte Algorithmus die Zustände des Präfixautomaten ...

(a) ... in der Reihenfolge einer Tiefensuche?

(b) ... in der Reihenfolge einer Breitensuche?

(c) ... weder in der Reihenfolge einer Tiefen- noch der Reihenfolge einer Breitensuche?

Lösung

In der Reihenfolge einer Tiefensuche: Von jedem Knoten aus wird zunächst (mittels des rekursiven Aufrufs) der "ersten" Kante zum nächsten Knoten nachgelaufen und dies solange, bis es nicht mehr "weitergeht" (d. h. bis kein weiterer neuer Zustand erzeut werden kann); genau wie bei der rekursiv implementierten Tiefensuche passiert beim rekursiven Abstieg ein Backtracking, das den eingeschlagenen Weg so weit zurückläuft, bis man auf einen Zustand trifft, von dem aus weitere Zustände erzeugt werden können.

Aufgabe 6.16

Erklären Sie einige der Einträge der Syntaxanalysetabelle (Tabelle **??**):

(a) Warum ist **Aktionstabelle**$[E_0, \texttt{(}] = \text{s6}$?

(b) Warum ist **Sprungtabelle**$[E_6, T] = 9$?

(c) Warum ist **Aktionstabelle**$[E_8, \texttt{)}] = \text{r5}$?

Lösung

(a) Es gilt **Aktionstabelle**$[E_0, \texttt{(}] = \text{s6}$, denn es gibt einen Zustandsübergang des Präfixautomaten von Zustand E_0 zu Zustand E_6, beschriftet mit `(`.

(b) Es gilt **Sprungtabelle**$[E_6, T] = 9$, denn es gibt einen Zustandsübergang des Präfixautomaten von Zustand E_6 nach Zustand E_9, beschriftet mit T.

(c) Es gilt **Aktionstabelle**$[E_8, \texttt{)}] = \text{r5}$, denn $\langle F \to \texttt{(}E\texttt{)}\ \texttt{.}\ \rangle \in E_8$ und `)` $\in \mathsf{FOLLOW}(F)$ und $\mathit{Grammatik.P}[5] = F \to \texttt{(}E\texttt{)}$.

Aufgabe 6.17

Schreiben Sie eine Funktion *printTab* als Methode der Klasse *Grammatik*, die die durch *tabCalc* berechnete Syntaxanalysetabelle in lesbarer Form ausgibt. Beispiel:

```
>>> print G.printTab()
     |    +    (    )    id   *    $    E'   E    T    F
 0   |         s6        s11                   1    9    10
 1   |    s2                       acc
 2   |         s6        s11                        3    10
 3   |    r1        r1        s4   r1
    ...                       ...
```

Lösung

Die folgende Methode *printTab* gibt die Syntaxanalysetabelle in gut lesbarer Form aus.

```
1 class Grammatik(object):
2   ...
3   def printTab(self):
4     def cntrs(strs): return ''.join(map(lambda s: s.center(5), strs))
5     lines = []
6     lines.append(['','|'] + self.T + self.V)
7     for i in range(len(self.items)):
8       lines.append([str(i),'|'] +
9                    ['' if a not in self.aktionTab[i]
```

```
                    else (('acc' if self.aktionTab[i][a][0]==ACCEPT
                    else 'r' if self.aktionTab[i][a][0]==REDUCE else 's') +
                  str(self.aktionTab[i][a][1])) for a in self.T ] +
                 ['' if Y not in self.sprungTab[i] else str(self.sprungTab[i][Y])
                   for Y in self.V ]
                )
    lines = map(cntrs, lines)
    print '\n'.join(lines)
```

Aufgabe 7.4

Schreiben Sie auf Basis der (bereits Python-artig formulierten) Formel (**??**) eine Python-Funktion, die die Verschiebetabelle eines als Parameter übergebenen Musters berechnet.

Lösung

Folgendes Listing zeigt eine Implementierung der Funktion P, die die Verschiebetabelle eines

```
def P(M):
  erg = []
  for i in range(len(M)):
    k = max([0] + [k for k in range(i) if M[:k]==M[i-k+1:i+1]])
    erg.append(k)
  return erg
```

Dies ist auch mit nur einer Listenkomprehension folgendermaßen zu implementieren:

```
def P2(M):
  return [max([0] + [k for k in range(i) if M[:k]==M[i-k+1:i+1]])
                for i in range(len(M))]
```

Aufgabe 7.5

Erstellen Sie die Verschiebetabelle für die folgenden Wörter:

(a) `ananas`

(b) `010011001001111`

(c) `ababcabab`

Lösung

(a) Die Verschiebetabelle für `ananas`:

i	: 0	1	2	3	4	5
$P[i]$	: 0	0	1	2	3	0
$M[i]$	: a	n	a	n	a	s

(b) Die Verschiebetabelle für `010011001001111`:

i	: 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$P[i]$	: 0	0	1	1	2	0	1	1	2	3	4	5	6	0	0
$M[i]$	: 0	1	0	0	1	1	0	0	1	0	0	1	1	1	1

(c) Die Verschiebetabelle für `ababcabab`:

i	: 0	1	2	3	4	5	6	7	8
$P[i]$	: 0	0	1	2	0	1	2	3	4
$M[i]$	: a	b	a	b	c	a	b	a	b

Aufgabe 7.7

Angenommen, wir suchen nach einem Muster M der Länge m in einem Text T der Länge n und angenommen alle mit $M[-1]$ verglichenen Zeichen kommen nicht im Muster vor – mit zunehmender Größe des verwendeten Alphabets wird dieser Fall natürlich wahrscheinlicher. Wie viele Suchschritte benötigt der Boyer-Moore-Algorithmus, bis er festgestellt hat, dass das Muster nicht im Text vorkommt?

Lösung

In diesem Fall benötigt der Boyer-Moore-Algorithmus $O(n/m)$ Schritte, also paradoxerweise umso weniger Schritte, je größer das Muster M ist. Grund: Die Bad-Character-Heuristik; bei jedem Mismatch dieser Art (wo das Mismatch-verursachende Zeichen nicht im Muster vorkommt) kann der Algorithmus um $m = len(M)$ Schritt nach vorne springen.

Aufgabe 7.8

Es stehe $\texttt{a}^n$ für die n-malige Wiederholung des Zeichens „a“. Wie viele Suchschritte benötigt der Boyer-Moore-Algorithmus um ...

(a) ... das Muster $\texttt{ba}^9$ (also das Muster `baaaaaaaaa`) im Text $\texttt{a}^{1000}$ (also einem Text bestehend aus 1000 as) zu finden?

(b) ... das Muster $\texttt{a}^9\texttt{b}$ (also das Muster `aaaaaaaaab`) im Text $\texttt{a}^{1000}$ zu finden?

(c) ... das Muster $\texttt{a}^9\texttt{b}$ (also das Muster `aaaaaaaaab`) im Text $\texttt{c}^{1000}$ zu finden?

Lösung

(a)

(b) Er benötigt in diesem Fall 1000 Suchschritte; jeder Vergleich des Musters mit dem Text beginnt mit dem Bad-Character „`a`“

(c) Er benötigt in diesem Fall $n/m = 1000/10 = 100$ Schritte; jeder Vergleich enthält einen Bad-Character, der nicht im Muster vorkommt; es kann also nach jedem Vergleich das Muster um 10 Stellen weitergeschoben werden.

Aufgabe 7.10

Beantworten Sie folgende Fragen zu Listing **??**:

(a) Erklären Sie die Zuweisung in Zeile 10; was würde passieren, wenn diese einfach „ *suffix* $= M[-j:]$“ heißen würde?

(b) Welchen Typ hat der Paramter *pat* im Aufruf der Funktion *unify* in Zeile 13? Welchen Typ hat der Parameter *suffix*?

(c) Es sei $M =$ `'ANPANMAN'`. Was sind die Werte von *suffix* und *mismatch* und in welchem Durchlauf bzw. welchen Durchläufen der „**for** *k*“-Schleife liefert dann der Aufruf von *unify* den Wert *True* zurück, wenn wir uns ...

1. ... im **for**-Schleifendurchlauf für j=1 befinden.
2. ... im **for**-Schleifendurchlauf für j=2 befinden.

Lösung

(a) Für den Fall j=0 (also Suffix-Länge 0) wäre dann *suffix* $= M[-0:]$, d. h. die Variable *suffix* würde also fälschlicherweise das gesamte Muster M enthalten.

(b) Die Variable *pat* ist eine Liste (von Zeichen), während die Variable *suffix* ein String ist; ein Vergleich mit *unify* ist trotzdem ohne Probleme möglich, denn beides sind Sequenzen, die mit mittels der *map*-Funktion (Zeile 5 in Listing **??**) elementweise auf Gleichheit überprüft werden. Gleichheit überprüft werden.

1. Im **for**-Schleifendurchlauf für j=1 gilt: *suffix*=`'N'`, *mismatch*=`'A'` und die „**for** *k*“-Schleife liefert nur für $k=-1$ (d. h. *pat*=[*DOT*,*DOT*]) den Wert *True* bei Aufruf von *unify* zurück.
2. Im **for**-Schleifendurchlauf für j=1 gilt: *suffix*=`'AN'`, *mismatch*=`'M'` und die „**for** *k*“-Schleife leifert für k=4 (d. h. *pat*=[`'P'`,`'A'`,`'N'`]) den Wert *True* zurück.

Aufgabe 7.11

Modifizieren Sie die in Listing **??** vorgestellte Funktion *boyerMoore* so, dass sie die Liste aller Matches des Musters M im Text T zurückliefert.

Lösung

```
1  def boyerMoore(T,M):
2    matches = []
3    delta1 = makedelta1(M)
4    delta2 = makedelta2(M)
5    m = len(M) ; n = len(T) ; i=m-1
6    while i < n:
7      i_old=i ; j=m-1
8      while j≥0 and T[i] == M[j]:
9        i-=1 ; j-=1
10     if j==-1:
11       matches.append(i+1)
12       i = i_old +1
13     else:
14       i = i_old + max(badChar(delta1,T[i],j), delta2[m-1-j])
15   return matches
```

Listing 9: Implementierung des Boyer-Moore-Algorithmus

Aufgabe 7.12

Gerade für den Fall, dass man mit einem bestimmten Muster komfortabel mehrere Suchen durchführen möchte, bietet sich eine objekt-orientierte Implementierung mittels einer Klasse *BoyerMoore* an, die man beispielweise folgendermaßen anwenden kann:

```
>>> p = BoyerMoore('kakaokaki')
>>> p.search(T1)
... .
>>> p.search(T2)
```

Implementieren Sie die Klasse *BoyerMoore*.

Lösung

Folgendermaßen lässt sich der Boyer-Moore-Algorithmus objekt-orientiert in einer Klasse kapseln:

```
1  class BoyerMoore(object):
2    def __init__ ( self ,M):
```

```
3       self.M = M
4       self.m = len(self.M)
5       self.delta1 = makedelta1(self.M)
6       self.delta2 = makedelta2(self.M)
7
8     def badChar(self,c,j):
9       try: return j - self.delta1[c]
10      except KeyError: return j+1
11
12    def search( self, T):
13      i=self.m-1
14      n=len(T)
15      while i < n:
16        i_old=i ; j=self.m-1
17        while j≥0 and T[i] == self.M[j]:
18          i-=1 ; j-=1
19        if j==-1:
20          print "Treffer: ",i+1
21          i = i_old +1
22        else:
23          i = i_old + max(self.badCharacter(T[i],j), self.delta2[self.m-1-j])
```

Aufgabe 7.13

Wäre auch die Konstante *sys.maxint* (aus dem Modul *sys*) ein sinnvoller Wert für M? Begründen Sie.

Lösung

In der Tat wäre es sinnvoll die Konstante *sys.maxint* als Wert von M zu verwenden. *sys.maxint* enthält die größte Ganzzahlkonstante, die das System unterstützt. Auf einem 64-Bit-Rechner wäre dies die Zahl $2^{64}/2-1$ – die andere Hälfte der 2^{64} vielen Ganzzahlen wird für die Repräsentation negativer Zahlen verwendet.

Aufgabe 7.14

Verwenden sie Pythons *timeit*-Modul, um die Laufzeiten der in Listing **??** und **??** gezeigten Funktionen *rollhash* und *rollhash2* zu vergleichen. Vergleichen Sie die Werte der timeit-Funktion für einen String S mit Länge 10, Länge 20 und Länge 50.

Lösung

Vergleich der Laufzeiten der beiden Hash-Implementierungen *rollhash* und *rollhash2*:

	$len(S)$=10	$len(S)$=20	$len(S)$=50
ohne Horner	2.93	19.50	72.16
mit Horner	2.27	4.02	9.21

Folgender Code wurde zur Durchführung des Vergleichs verwendet:

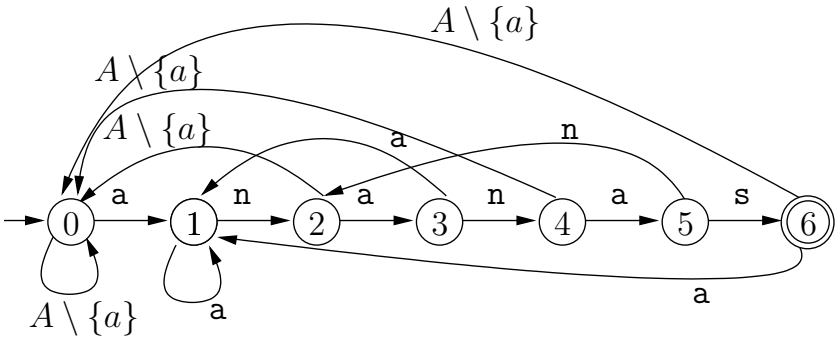
```
# Erzeugen von Beispielstrings der Laengen 10, 20 und 50
M10 = Ms[0]
M20 = Ms[0] + Ms[1]
M30 = Ms[0] + Ms[1] + Ms[2]
M50 = M30 + M20

from timeit import Timer
t10_1 = Timer('rollhash(M10)','from __main__ import M10,rollhash')
t10_2 = Timer('rollhash2(M10)','from __main__ import M10,rollhash2')

t20_1 = Timer('rollhash(M20)','from __main__ import M20,rollhash')
t20_2 = Timer('rollhash2(M20)','from __main__ import M20,rollhash2')

t50_1 = Timer('rollhash(M50)','from __main__ import M50,rollhash')
t50_2 = Timer('rollhash2(M50)','from __main__ import M50,rollhash2')
```

Aufgabe 7.15

Erstellen Sie einen *deterministischen* endlichen Automaten, der Vorkommen des Wortes `ananas` erkennt.

Lösung

Folgender deterministischer endlicher Automat erkennt des Wort `ananas`:



Aufgabe 7.17

Führen Sie einen direkten Performance-Vergleich der bisher vorgestellten String-Algorithmen durch.

- Der Vergleich sollte mit einem relativ kurzen Muster (10 Zeichen) und einem relativ langen Muster (50 Zeichen) auf einer relativ kleinen Datenmenge (1000 Zeichen) und einer relativ großen Datenmenge (ca. 1 Million Zeichen) durchgeführt werden.
- Der Vergleich sollte mit dem naiven String-Matching-Algorithmus, dem Knuth-Morris-Pratt-Algorithmus, dem Boyer-Moore-Algorithmus, dem Rabin-Karp-Algorithmus und dem Shift-Or-Algorithmus durchgeführt werden.

Lösung

Folgendes Skript führt den gewünschten Performance-Vergleich aus:

```
1 for alg in ['shiftOr','rabinKarp1','boyerMoore','kmp','match']:
2   for muster in ['M_kurz','M_lang']:
3     for text in ['T_kurz','T_lang']:
4       algT = Timer(alg+'('+muster+','+text+')','from __main__ import '+alg+','+muste
5       print alg,muster,text,': ',algT.timeit(number=1)
```

... und die folgenden Ergebnisse wurde erzeugt:

```
shiftOr M_kurz T_kurz :  0.000139951705933
shiftOr M_kurz T_lang :  0.219347953796
shiftOr M_lang T_kurz :  0.00033712387085
shiftOr M_lang T_lang :  0.404415130615
rabinKarp1 M_kurz T_kurz :  0.000667810440063
rabinKarp1 M_kurz T_lang :  2.67133402824
rabinKarp1 M_lang T_kurz :  0.000187873840332
rabinKarp1 M_lang T_lang :  11.6732189655
search M_kurz T_kurz :  0.00100493431091
search M_kurz T_lang :  0.136551856995
search M_lang T_kurz :  28.1512029171
search M_lang T_lang :  28.3747599125
kmp2 M_kurz T_kurz :  0.000131130218506
kmp2 M_kurz T_lang :  0.377729892731
kmp2 M_lang T_kurz :  0.061005115509
kmp2 M_lang T_lang :  0.443845033646
match M_kurz T_kurz :  0.000361919403076
match M_kurz T_lang :  1.22638702393
match M_lang T_kurz :  0.00187397003174
match M_lang T_lang :  7.03574514389
```

Aufgabe 8.1

Geben Sie eine direkt rekursive Implementierung einer Lösung des Travelling-Salesman-Problems in Python an, basierend auf Formel (**??**).

Lösung

Folgendes Listing zeigt eine (sehr ineffiziente) rekursive Implementierung einer Lösung des Travelling-Salesman-Problems, die direkt auf dem in Formel (**??**) formulierten für das TSP-geltenden Bellmanschen Optimalitätsprinzip basiert.

```
1 def tsp2Rek(graph): # direkt rekursive Implementierung
2   def T(i,S):
3       if len(S) == 0: return graph.w(i,1)
4       return min( graph.w(i,j) + T(j, diff(S,[j])) for j in S )
5   return T(1,range(2,graph.numNodes+1))
```

Listing 10: Rekursive Implementierung einer Lösung des Travelling-Salesman-Problems.

Der Grund, warum diese Lösung so ineffizient ist: $T(i,S)$ wird für ein festes i und ein festes S viele Male aufgerufen; i. A. umso öfter, je größer $|S|$ ist.

Aufgabe 8.2

Modifizieren Sie den in Listing **??** gezeigten Algorithmus so, dass er – zusätzlich zur Länge der kürzesten Route – die kürzeste Route selbst als Liste von zu besuchenden Knoten zurückliefert.

Lösung

Das folgende Listing 11 zeigt die Modifikation des in Listing **??** gezeigten Algorithmus so, dass zusätzlich zur Länge der kürzesten Route auch die Route selbst als Liste zurückgeliefert wird.

```
1  def tspPath(graph): # Der kürzeste Weg wird auch zurückgegeben.
2    n = graph.numNodes
3    T = {}
4    for i in range(1,n+1): T[(i,())] = (graph.w(i,1), i)
5    for k in range(1,n-1):
6      for S in choice(range(2,n+1),k):
7        S = tuple(S)
8        for i in diff(range(2,n+1),S):
9          minG = min( (graph.w(i,j) + T[(j,diff(S,[j]))][0], j)  for j in S )
10         T[(i,S)] = minG
```

```
11  S=tuple(range(2,n+1))
12  (m,i)=min( (graph.w(1,j)+T[(j,diff(S,[j]))][0],j) for j in range(2,n+1) )
13  erg=[1,i] ; S = diff(S,[i])
14  while S≠():
15    i = T[(i,S)][1] ; erg.append(i) ; S = diff(S,[i])
16  return (m,erg+[1])
```

Listing 11: Lösung des Travelling-Salesman-Problems, die zusätzlich zum Wert einer minimalen Tour die Tour selbst als Liste zurückliefert

Um die eigentliche Route zu speichen, merkt sich der Algorithmus bei jeder Berechnung einer kürzesten „Teil"-Tour $T(i, S)$ nicht nur den Wert dieser Tour sondern auch den Knoten j, der von i aus als erstes besucht werden muss, also:

$$T(i, S) = \min_{j \in S}(w(i, j) + T(j, S \setminus \{j\}), j)$$

(die Minimums-Berechnung ist hier bezogen auf die erste Komponente der erzeugten Tupel).
Nach der finalen Berechnung der kürzesten Tour in Zeile 12, läuft der Algorithmus genau diese in der Tabelle zusätzlich protokollierten Knoten zurück und speichert diese in der Ergebnisliste; dies geschieht in der **while**-Schleife ab Zeile 14.

Aufgabe 8.3

Vergleichen Sie die Implementierung in Listing **??**, die eine Lösung des TSP-Problems durch Ausprobieren aller Möglichkeiten berechnet, mit der Implementierung aus Listing **??**, die Dynamische Programmierung verwendet.

(a) Zur Berechnung der in Abbildung **??** gezeigten Lösung, die kürzeste Rundtour durch die 20 größten Städte Deutschlands zu finden, hat der *tsp*-Algorithmus aus Listing **??** auf dem Rechner des Autors etwa 4 Minuten benötigt. Schätzen Sie ab, wie lange der Algorithmus aus Listing **??** zur Berechnung dieser Lösung benötigen würde.

(b) Wie viel mal mehr Schritte benötigt der Algorithmus aus Listing **??** wie der auf Dynamsicher Programmierung basierende Algorithmus um eine Rundreise durch n Städte zu berechnen?

Lösung

Die Berechnung einer kürzesten Tour über n Städte benötigt

$$\frac{2^{n-1} \cdot (n/2)^2}{2^{19}/100}$$

mal mehr Schritte als die Berechnung einer kürzesten Tour über 20 Städte. Davon ausgehend, dass die Berechnung der kürzesten Tour über 20 Städte etwas 4 Minuten benötigt,

erhalten wir folgende Berechnungszeiten:

Anzahl Städte	$n = 20$	$n = 30$	$n = 40$	$n = 50$	$n = 60$
Faktor	1×	2304×	4194304×	6710886400×	9895604649984×
Laufzeit (ca.)	4 min	6 Tage	31 Jahre	51072 Jahre	75 Mio Jahre

Hier „spürt“ man die exponentielle Laufzeit: bei einer konstanten Erhöhung von n um 10 vergrößert sich die Zahl, die die Laufzeit repräsentiert, um jeweils 3-4 Stellen!

Sehr schnell (schneller jedenfalls als alles in den Taschenrechner einzutippen) bekommt man die entsprechenden Zahlen mit Hilfe von Python:

```
>>> def n(x): return 2**(x-1) *(x/2)**2 # Anzahl Schritte für n Städte
>>> map(lambda x: n(x)/n(20), [20,30,40,50,60])
>>> [1, 2304L, 4194304L, 6710886400L, 9895604649984L]
```

Aufgabe 8.5

Implementieren Sie die Nearest-Neighbor-Heuristik für das Travelling-Salesman-Problem und testen Sie diese durch Berechnung der kürzesten Tour durch die ...

(a) ... größten 20 deutschen Städte.

(b) ... größten 40 deutschen Städte.

Hinweis: Die einfachste Möglichkeit, sich einen Graphen zu erzeugen, der die 20 bzw. 40 größten deutschen Städte enthält, besteht in der Verwendung des Python-Moduls *pygeodb*. Mittels *pygeodb.distance* erhält man etwa den Abstandswert zweier Städte.

Lösung

Folgendes Listing zeigt die Implementierung der Nearest-Neighbor-Heuristik für das TSP.

```
1 def tspNearestNeighbor(graph,j):
2   tour = [j]
3   while len(tour)<graph.numNodes:
4       (w,j) = min([ (graph.w(j,i), i) for i in range(1,graph.numNodes+1) if (i not in tour) ])
5       tour.append(j)
6   tour.append(tour[0]) # Wieder zurueck zum Ausgangspunkt
7   return pathVal(graph, tour), tour
```

Listing 12: Berechnung einer kurzen Rundtour mit Hilfe der Nearest-Neighbor-Heuristik.

Die **while**-Schleife ab Zeile 3 läuft so lange, bis die Tour aus n (= Anzahl der Knoten im Graphen) Städten besteht. Die **for**-Schleife in der Listenkomprehension in Zeile 4

läuft immer über alle noch nicht in der Tour befindlichen Knoten; es wird immer der Knoten i mit dem zum aktuellen Knoten j minimalen Abstand in die Tour eingefügt. Nachdem sich alle Knoten in der Tour befinden wird in Zeile 6 die Tour wieder zum Ausgangsknoten zurückgeführt und schließlich die Länge der Tour inklusive der Tour selbst zurückgeliefert.

(a) Der Implementierung der Nearest-Neighbor-Heuristik wird zusätzlich ein Startknoten übergeben – die Güte der ermittelten Tour hängt stark von der Wahl des Startwertes ab. Eine Beispielanwendung unter Verwendung eines Entfernungsgraphen *st* der die 20 größten Städte Deutschlands verbindet ergibt:

```
>>> tspNearestNeighbor(st,1)  # 1 ≙ "Berlin"
>>> (2512, [1, 12, 13, 14, 3, 6, 20, 5, 19, 4, 7, 15, 9, 16, 8, 17, 18, 10, 2, 11, 1])
```

Noch günstiger ist es, den Startwert zu ermitteln, mit dem man durch die Nearest-Neighbor-Heuristik eine möglichst kurze Tour bekommt. In Pyhton lässt sich dies folgendermaßen bewerkstelligen:

```
>>> min( (tspNearestNeighbor(st, i)[0], i) for i in range(1, st.numNodes+1))
>>> (2494,2)
```

Die beste Nearest-Neighbor-Tour erhält man also, wenn man bei Knoten 2 (das ist in diesem Falle `'Hamburg'`) beginnt.

(b) Übrigens erhält man mit der Nearest-Neighbor-Statisik für die größten 40 deutschen Städte eine Rundtour der Länge 5117 Kilometer (beginnend mit Fulda oder Würzburg).

Aufgabe 8.6

Implementieren Sie die Nearest-Insertion-Heuristik zum Finden einer möglichst optimalen Lösung des Travelling-Salesman-Problems.

Lösung

Das folgende Listing zeigt die Implementiernug der Nearest-Insertion-Heuristik:

```
1 def tspNearestIns(graph):
2   n = graph.numNodes
3   def distTour(tour,x): # liefert Abstand und Knoten
4     return min([graph.w(i,x) for i in tour])
5   (w,a,b) = min([(graph.w(i,j), i, j) for i in range(1,n+1) for j in range(1,n+1) if i≠j])
6   tour = [a,b]
7   while len(tour)<n:
8     v = min([(distTour(tour,i), i) for i in range(1,n+1) if i not in tour])[1]
9     pos = min([ (graph.w(tour[i],v) + graph.w(v,tour[i+1]) -graph.w(tour[i],tour[i+1]), i)
```

```
10                                  for i in range(0,len(tour)-1)  ])[1]
11      tour.insert(pos+1,v)
12    tour = tour + [tour[0]]   # Rundtour daraus machen
13    return pathVal(graph,tour)/1000, tour
```

Im Gegensatz zur Random-Insertion-Heuristik wird jetzt derjenige Knoten mit dem kleinsten Abstand zur momentanen Tour bestimmt – dies geschieht in der Listenkomprehension in Zeile 8. Für alle Knoten i mit $i \notin tour$ wird ein Tupel mit Abstandswert und Knotennummer erzeugt. Die Mimimumsberechnung liefert nun Abstandswert und Knoten mit minimalem Abstand zur Tour. Über die Indiziertung $min(\ldots)[1]$ erhält man den gewünschten Knoten.

Die in den Zeilen 3 und 4 definierte lokale Funktion berechnet den Abstand eines Knotens zu einer Tour.

Aufgabe 8.7

Implementieren Sie die Farthest-Insertion-Heuristik zum Finden einer möglichst optimalen Lösung des Travelling-Salesman-Problems.

Lösung

Die Implementierung der Farthest-Insertion-Heuristik ist – bis auf die Verwendung der *max*-Funktion in Zeile 8 – identisch mit der Implementierung der Nearest-Insertion-Heuristik:

```
1  def tspFarthestIns(graph):
2    n = graph.numNodes
3    def distTour(tour,x):
4      return min([graph.w(i,x) for i in tour])
5    (w,a,b) = min([(graph.w(i,j), i, j) for i in range(1,n+1) for j in range(1,n+1) if i≠j])
6    tour = [a,b]
7    while len(tour)<n:
8      v = max([(distTour(tour,i), i) for i in range(1,n+1) if i not in tour])[1]
9      pos = min([ (graph.w(tour[i],v) + graph.w(v,tour[i+1]) -graph.w(tour[i],tour[i+1]), i)
10                                  for i in range(0,len(tour)-1)  ])[1]
11     tour.insert(pos+1,v)
12   tour = tour + [tour[0]]
13   return pathVal(graph,tour)/1000, tour
```

Aufgabe 8.8

Vergleichen Sie die Güte der gefundenen Lösungen durch die in Listing **??** gezeigte Implementierung der Random-Insertion mit den durch ...

- ... Nearest-Insertion
- ... Farthest-Insertion

... gefundenen Lösungen.

Lösung

Erstaunlicherweise liefert die Nearest-Insertion-Heuristik deutlich schlechtere Ergebisse als die Random-Insertion-Heuristik. Die Random-Insertion- und Farthes-Insertion-Heuristik sind etwa gleich auf. Folgende Tabelle zeigt die Werte der berechneten Touren über die 100 größten deutschen Städte für die verschiedenen Heuristiken (der Werte für die Random-Insertion-Heuristik ist ein Durchschnittswert über 100 Berechnungen.

Nearest-Neigh.	Nearest-Neigh-Min	Nearest-Ins.	Random-Ins.	Farthest-Ins
5078 km	4544 km	5009 km	4500 km	4513 km

Random-Insertion liefert übrigens nicht nur die besten Ergebnisse, es ist auch (abgesehen von der einfachen Nearest-Neighbor-Heuristik) die schnellste Methode. Die mit „Nearest-Neigh-Min“ bezeichnete Spalte ist die Nearest-Neighbor-Heuristik mit dem optimalen Anfangsknoten.

Die Abbildungen 1 und 2 zeigen jeweils eine Tour durch die größten 40 Städte Deutschlands berechnet mit der Nearest-Insertion-Heuristik bzw. der Farthest-Insertion-Heuristik.

Aufgabe 8.9

Was die Laufzeit betrifft, kann die in Listing **??** gezeigte Implementierung der Tourverschmelzung verbessert werden. Anstatt die optimalen Verschmelzungs-Knoten jedesmal neu zu berechnen – wie in den Zeilen 9 und 10 in Listing **??** – kann man sich jeweils die optimalen Nachbarn der Anfangs- und Endknoten einer Teiltour merken und – nach einer Verschmelzung – gegebenenfalls anpassen.
Entwerfen Sie eine entsprechend optimierte Version der in Listing **??** gezeigten Implementierung und analysieren Sie, welche Laufzeit der Algoritmus nach diese Optimierung hat.

Lösung

Implementierung dieser Optimierung:

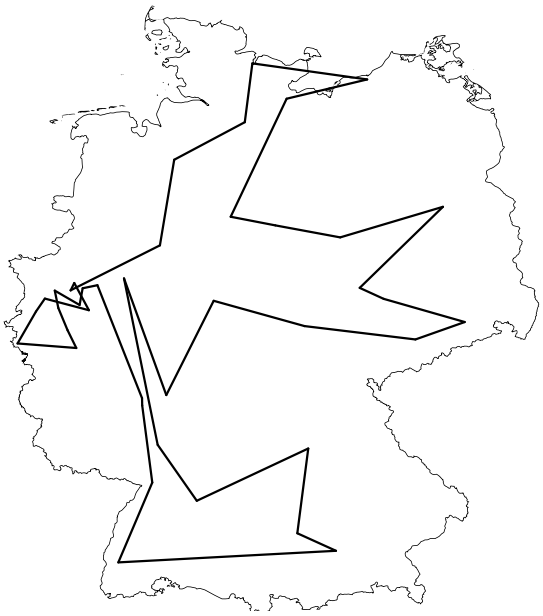


Abbildung 1: Tour durch die größten 40 Städte Deutschlands, gefunden mittels der Nearest-Insertion-Heuristik.

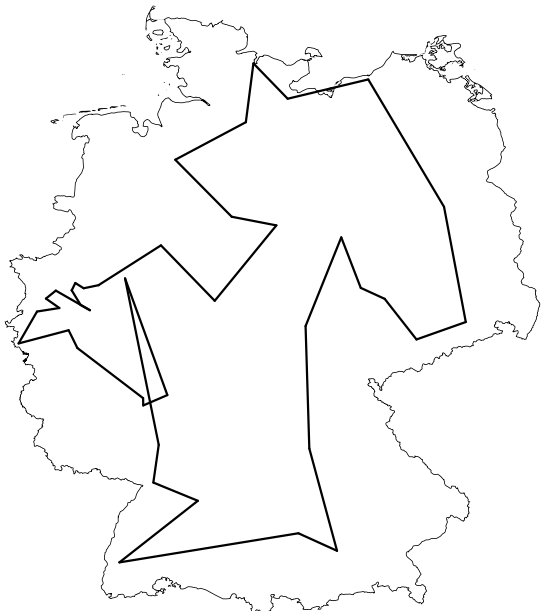


Abbildung 2: Tour durch die größten 40 Städte Deutschlands, gefunden mittels der Farthest-Insertion-Heuristik.

Aufgabe 8.10

Implementieren Sie die Funktion *all2_5Cuts*(n), die alle Spezifikationen aller möglichen Löschungen dreier Kanten erzeugt. Beispiel-Anwendungen:

```
>>> all2_5Cuts(10)
>>> [ ((0,1),(3,4,5)),    ((0,1),(4,5,6)),    ... ,    ((5,6,7),(8,9)) ]
```

Lösung

Durch Listenkomprehensionen kann man leicht alle Spezifikationen für Löschungen dreier Kanten für das 2.5-Opt-Verfahren erzeugen.

```
1 def all2_5Cuts(n):
2   return [((j,j+1),(i,i+1,i+2)) for i in range(0,n-2) for j in range(0,i-2)] +\
3          [((i,i+1,i+2),(j,j+1)) for i in range(0,n-2) for j in range(i+3,n-1)]
```

Die „erste“ Listenkomprehension aus Zeile 2 erzeugt alle Möglichkeiten, die beiden zu löschenden benachbarten Kanten *nach* der einen Kante zu platzieren; die „zweite“ Listenkomprehension aus Zeile 3 erzeugt alle Möglichkeiten, die beiden zu löschenden benach-

barten Kanten *vor* der einen Kante zu platzieren. Die mittels des +-Operators erzeugte Vereinigung dieser beiden entsprechenden Mengen bilden alle möglichen Löschungen für die 2.5-Opt-Heuristik.

Aufgabe 8.11

(a) Verwenden Sie statt der *map*-Funktion in Zeile 2 in Listing **??** eine Listenkomprehension.

(b) Schreiben Sie die in Listing **??** gezeigte Funktion *tsp2_5Opt* so um, dass der Funktionskörper lediglich aus einem **return**-Statement besteht.

Lösung

Das folgende Listing zeigt die Implementierung der 2.5-Opt-Heuristik in nur einem **return**-Statement – allerdings ist diese Implementierung langsamer als die in Listing **??** gezeigte, denn in jeder Iteration wird *crossTour* zweimal aufgerufen.

```
1 def tsp2_5Opt(graph,tour):
2   return min( [(pathVal(graph,crossTour2_5(tour,crTr)), crossTour2_5(tour,crTr))
3                         for crTr in all2_5Cuts(len(tour)) ]
```

Aufgabe 8.14

Wie viele Möglichkeiten gibt es eine durch Löschung von k Kanten zerfallene Tour wieder neu zu verbinden? Geben Sie eine entsprechende von k abhängige Formel an.

Lösung

Sei A_k die Anzahl der Möglichkeiten der Neuverbindung einer durch Löschung von k disjunkten Kanten entstandenen Tour. Wir gehen davon aus, dass die Kanten (v_{i_0}, v_{i_0+1}), (v_{i_1}, v_{i_1+1}), $\ldots$, $(v_{i_{k-1}}, v_{i_{k-1}+1})$ mit $i_0 < i_1 + 1 \wedge i_1 < i_2 + 1 \wedge \ldots \wedge i_{k-1} < i_k + 1$ gelöscht wurden. Wir lassen als Neuverbindung auch Kanten der ursprünglichen Tour zu und beginnen damit, Knoten v_{i_0} neuzuverbinden: Für v_{i_0} gibt es $2k-2$ mögliche Neuverbindungen (außer v_{i_0} sind zu Beginn $2k-1$ Knoten „frei“, d. h. haben Grad 1; Knoten $v_{k-1}+1$ kann jedoch nicht gewählt werden, sonst zerfällt die Tour in mindestens zwei Stücke). Durch eine Neuverbindung gibt es im zweiten Schritt zwei „freie“ Knoten weniger, d. h. jetzt kann aus $2k-4$ möglichen Neuverbindungen gewählt werden, usw. Insgesamt Ergibt sich also

$$A_k = (2k-2) \cdot (2k-4) \cdot \ldots \cdot 2 = \prod_{i=1}^{k-1} 2k - 2i$$

Aufgabe 8.15

Vor allem wenn k relativ groß ist (etwa $k > 5$), ist es nicht immer sinnvoll sich systematisch alle Kreuztouren generieren zu lassen; in diesen Fällen tut man besser daran, sich zufällig *eine* der vielen möglichen Kreuztouren auszuwählen. Implementieren Sie eine entsprechende Python-Funktion *randCross*, die – genau wie die Funktion *allCrosses* aus Listing **??** – eine Liste der fehlenden Tourkanten als Argument übergeben bekommt und eine zufällig ausgewählte Kreuz-Tour zurückliefert.

Lösung

Folgendes Listing zeigt die Implementierung der Funktion *randCrosses*, die eine zufällige Kreuztour einer durch Löschung von k Kanten zerfallenen Tour zurückliefert.

```
1  def randomCross(i):
2    crossTour = []
3    while len(i)>1:
4      p = random.randint(0,len(i)-1)
5      if p == len(i)-1: forward = False
6      elif  p == 0: forward = True
7      else: forward = random.choice([False, True])
8      if forward:
9        crossTour.append((i[p][1], i[p+1][0]))
10       i = i[:p] + [(i[p][0], i[p+1][1])] + i[p+2:]
11     else:
12       crossTour.append((i[p][0], i[p-1][1]))
13       i = i[:p-1] + [(i[p-1][0],i[p][1])] + i[p+1:]
14   return crossTour
```

p ist der Index des zufällig gewählten Kantentupels; *forward* ist ein zufällig gewählte Variable, die angibt, ob durch die neu einzufügende Kante eine Vorwärts- oder Rückwärtsteiltour gegangen werden soll; für weitere Erläuterungen sei auf die entsprechenden Erläuterungen zu Listing **??** auf Seite **??** verwiesen.

Aufgabe 8.16

Implementieren Sie die k-Opt-Heuristik folgendermaßen:

(a) Schreiben Sie zunächst eine Funktion *randCut*(n,k), die aus einer Tour mit n Knoten zufällig k disjunkte Kanten auswählt und die Anfangsknoten dieser Kanten zurückliefert.

```
>>> randCut(100,5)
>>> [16, 30, 73, 84, 99]
```

(b) Schreiben Sie eine Funktion *kOpt*(*graph*,*k*,*m*), die die kOpt-Heuristik implementiert. Für $j = k, k-1, \ldots, 2$ werden jeweils n-mal zufällig j zu löschende Kanten gewählt; aus dieser entsprechend zerfallenen Tour wird die kürzeste Kreuztour gewählt.

Lösung

(a) Eine sehr einfache – wenn auch nicht die aller-effizienteste – Möglichkeit sich zufällig k disjunkte Tourkanten auszuwählen, besteht darin, sich solange zufällig k Zahlen aus $\{0, \ldots, n-1\}$ auszuwählen, bis diese Zahlen alle einen Abstand von mindestens 2 haben:

```
1 def randCut(n,k):
2   while True:
3     l = sorted(random.sample(range(n),k))
4     if False not in map(lambda x,y: x+2≤y, l[:-1], l[1:]): return l
```

In Zeile 3 werden mittels *random.sample* zufällig k Zahlen aus $\{0, \ldots, n-1\}$ ausgewählt. Die *map*-Funktion in Zeile 4 prüft alle benachbarten Werte dieser Liste daraufhin, ob sie einen Abstand von mindestens 2 haben. Nur falls dies für alle Werte zutrifft, werden diese k Zahlen zurückgeliefert – andernfalls werden wiederum k zufällige Zahlen ausgewählt, usw.

(b) Die kOpt-Heuristik kann nun folgendermaßen implementiert werden:

```
1 def VOpt(graph,tspHeuristik,k,m):
2   n = graph.numNodes
3   tour = tspHeuristik(graph)[1]
4   for j in range(k,1,-1):
5     for i in range(m):
6       cut = randCut(n,j)
7       tour = min(map(lambda x: (pathVal(graph,x),x), allCrossTours(tour,cut)))[1]
8   return tour
```

In Zeile 3 wird mittels einer einfachen (als Parameter übergebenen) Heuristik eine Ausgangstour erzeugt. Für jedes $j \in \{k, \ldots 2\}$ werden dann m-mal zufällig j zu löschende Kanten ausgewählt. In Zeile 7 wird für diesen Schnitt diejenige Kreuztour mit minimaler Länge gesucht.

Aufgabe 8.17

Wir wollen eine Variante der kOpt-Heuristik implementieren, die gewährleistet, dass *alle* Kreuztouren, *aller* möglichen Schnitte mit in Betracht gezogen werden.

(a) Implementieren Sie eine Funktion *allCuts*(n,k), die die Liste aller möglichen Löschungen von k Kanten aus einer Tour mit n Knoten erzeugt.

(b) Implementieren Sie eine Funktion *kOptAll*(*graph*,k), die die k-Opt-Heuristik implementiert und hierbei tatsächlich alle Möglichkeiten durchspielt.

Lösung

Am einfachsten – zumindest dann, wenn man im Programmieren rekursiver Funktionen geübt ist – lässt sich die Liste aller Mengen von n zu löschenden Kanten folgendermaßen erzeugen:

```
1 def allCuts(anf,ende,n):
2     ret = []
3     if n==1: return [[i] for i in range(anf,ende+1)]
4     for i in range(anf,(ende+1)-2*(n-1)):
5         ret += [[i]+c for c in allCuts(i+2,ende,n-1)]
6     return ret
```

In Zeile 3 befindet sich der Rekursionsabbruch: Die Liste aller 1-elementigen zu löschenden Kanten ist gerade [[*anf*], [*anf*+1], ... , [*ende*]]. Die Zeilen 4 und 5 implementieren den Rekursionsschritt. Die Variable i durchläuft hierbei alle Möglichkeiten, die „erste“ zu löschende Kante (d. h. diejenige Kante mit kleinstem Index) zu wählen. Man beachte dass i höchstens den Wert $ende - 2(n-1)$ haben darf (sonst gibt es keinen Platz mehr für die restlichen zu wählenden Kanten).

Aufgabe 8.18

Implementieren Sie die in Zeile 9 in Listing **??** verwendete Funktion *randomWalk* folgendermaßen: *randomWalk* soll mit vorhandenen Kanten versuchen eine zufällige Tour zu konstruieren. Sollte es nicht mehr „weitergehen“, weil alle Nachbarn des aktuellen Knotens schon besucht wurden, dann sollte *randomWalk* zurücksetzen und bei einem vorherigen Knoten eine andere Alternative wählen (ein solches Zurücksetzen nennt man auch *Backtracking*).

Lösung

Die Konstruktion einer Rundtour auf einem Graphen *graph*, beginnend bei Knoten v, kann folgendermaßen implementiert werden:

```
1 def randomWalk(graph,v,start=None,visited=[]):
2   if not start: start = v
3   if len(visited)+1 == graph.numNodes: return visited+[v]+[start]
4   neighbors = [u for u in graph.G(v) if u not in visited]
5   random.shuffle(neighbors)
```

```
6   for u in neighbors:
7     walk = randomWalk(graph, u, start, visited + [v])
8     if walk: return walk
9   return False
```

Im dritten Parameter, den wir in Zeile 2 setzen, merken wir uns den Startknoten um beim Rekursionsabbruch in Zeile 3 aus den gefundenen Knoten eine Rundtour machen zu können. In der Variablen *neighbors* suchen wir uns diejenigen Nachbarn des aktuellen Knotens v, die bisher noch nicht besucht wurden und verwenden das *random.shuffle*-Kommando, um diese Nachbarn in eine zufällige Reihenfolge zu bringen. Die **for**-Schleife in Zeile 6 zusammen mit dem rekursiven Aufruf von *randomWalk* in Zeile 7 realisieren das Backtracking : es wird der Reihe nach mit allen Nachbarn von v versucht eine Rundtour zu konstruieren; sollte dies scheitern, d. h. sollte *randomWalk* für alle Nachbarn von v den Wert *False* leifern, so wird am Ende auch *False* zurückgeliefert.

Aufgabe 8.19

Der in Listing **??** gezeigte genetische Algorithmus für das Travelling-Salesman-Problem weist folgende Schwäche auf: Die Populationen tendieren dazu, über die Zeit (nach etwa 5 Generationen) genetisch zu verarmen – in diesem Fall heißt das: viele der erzeugten Individuen sind gleich.

(a) Passen Sie den Algorithmus so an, dass sichergestellt wird, dass eine Population keine identischen Individuen enthält.

(b) Man stellt jedoch schnell fest: Der Algorithmus „schafft“ es nach einigen Generationen grundsätzlich nicht mehr, neuartige Individuen hervorzubringen. Passen Sie den Algorithmus so an, dass maximal 50-mal versucht wird ein neues Individuum hervorzubringen – danach wird einfach ein schon vorhandenes Individuum der Population hinzugefügt.

Lösung

(a) Folgendermaßen kann der Algorithmus aus Listing **??** angepasst werden, damit keine Population (außer eventuell der initialen) identische Individuen enthält:

```
1  def tspGen(graph, p, g):
2    pop = sorted([tspRandIns(graph) for i in range(p)])
3    for i in range(g):
4      print [s[0] for s in pop]
5      newPop = pop[:p/3]
6      while len(newPop)<5*len(pop):
7        tours = random.sample(pop,2)
8        childTour = edgeCrossOver(graph, tours[0][1], tours[1][1])
9        if childTour not in [s[1] for s in newPop] or misses>20:
```

```
10            newPop.append((pathVal(graph,childTour)/1000, childTour))
11        pop = sorted(newPop)[:p]
12    return pop
```

Neu ist hier lediglich die **if**-Abfrage in Zeile 9: ein neues Individuum wird nur dann in die Population aufgenommen, wenn es nicht bereits existiert.

(b) Folgendermaßen kann der Algorithmus aus dem in Teilaufgabe (a) gezeigten Listing angepasst werden, damit maximal 50-mal versucht wird ein neues Individuum hervorzubringen – danach wird einfach ein schon vorhandenes *newPop* eingefügt.

```
1  def tspGen(graph, p, g):
2      pop = sorted([tspRandIns(graph) for i in range(p)])
3      misses = 0
4      for i in range(g):
5          print [s[0] for s in pop]
6          newPop = pop[:p/3] # das beste drittel ueberlebt
7          while len(newPop)<5*len(pop):
8              tours = random.sample(pop,2)
9              childTour = edgeCrossOver(graph, tours[0][1], tours[1][1])
10             if childTour not in [s[1] for s in newPop] or misses>20:
11                 newPop.append((pathVal(graph,childTour)/1000, childTour))
12             else: misses += 1
13             misses = 0
14         pop = sorted(newPop)[:p]
15     return pop
```

Mittels der Variablen *misses* wird sichergestellt, dass höchstens 20 mal versucht wird, ein neues Individuum zu erzeugen.

Aufgabe 8.20

Der Algorithmus in Listing **??** verwendet für zur Implementierung eines genetischen Algorithmus das Kanten-Cross-Over als Reproduktionsart. Implementieren Sie eine Variante, die stattdessen das Knoten-Cross-Over verwendet und vergleichen Sie die Qualitäten der Ergebnisse für die beiden Reproduktionstechniken.

Lösung

Aufgabe 8.21

Implementieren Sie die Funktion *chooseIndex*, die eine Liste von Zahlen $[x_1, \ldots x_n]$ übergeben bekommt und mit Wahrscheinlichkeit p_i die Zahl i zurückliefert, wobei

$$p_i = \frac{x_i}{\sum_{k=1}^{n} x_k}$$

Lösung

Die Funtion *chooseIndex* kann folgendermaßen implementiert werden:

```
1  def chooseIndex(lst):
2      ''' Waehlt Index gemaess Gewichtung der Werte in lst '''
3      s = float(sum(lst)) ; i=0
4      lstNorm = []
5      r = random() ; j=0
6      if s==0: return choice(range(len(lst)))
7      for l in lst:
8          i += l/s
9          if i<r: j+=1
10         else: break
11     return j
```

Jeder Eintrag der Liste *lst* wird hierbei normiert, indem mit s dividiert wird.

Aufgabe 8.22

In Zeile 5 in Listing **??** werden die Touren ihrer Länge nach sortiert, um schließlich die kürzeste Tour zurückzuliefern, die in diesem Zyklus von einer Ameise gelaufen wurde.

(a) Es gibt jedoch eine schnellere Methode – zumindest was die asymptotische Laufzeit betrifft – die kürzeste Tour zu erhalten. Welche?

(b) Implementieren Sie mit Hilfe dieser Methode eine schnellere Variante von *acoCylceH*.

(c) Führen mit Hilfe von Pythons *timeit*-Modul Laufzeitmessungen, um zu prüfen, ob *acoCycleH* tatsächlich performanter ist als *acoCylce*.

Lösung

(a) Es gibt eine – zumindest was die asymptotische Laufzeit betrifft – schnellere Methode als das Sortieren, nämlich der Aufbau eines Heaps. Während das Sortieren eine

Laufzeit von $O(n \cdot \log n)$ hat, benötigt der Aufbau eines Heaps lediglich $O(n)$ Schritte.

(b) Folgende Alternative Implementierung verwendet einen Heap:

```
1 def acoCycleH(graph):
2   tours = [ant(graph) for _ in range(M)]
3   tours2 = [(tl,t) for t,tl in tours] # umtupeln
4   vapourize(graph)
5   for (t,tl) in tours: adapt(graph,t,tl)
6   heapify(tours2)
7   #print "Kuerzeste Tour ist",tours2 [0][0]," lang"
8   return tours2[0][0]
```

(c) Laufzeitmessungen:

```
>>> from timeit import Timer
>>> t1 = Timer("acoCycle(st20Ph)", "from __main__ import acoCycle, st20Ph")
>>> t2 = Timer("acoCycleH(st20Ph)", "from __main__ import acoCycleH, st20Ph")
>>> t1.timeit(number=100)
2.300879955291748
>>> t2.timeit(number=100)
2.3378729820251465
```

Diese „Optimierung“ bringt also noch nicht den erwünschten Erfolg; das ist wohl der intelligenten Implementierung von *sort*() zu verdanken.

Aufgabe 8.23

Wenden Sie den „verbesserten“ Ameisenalgorithmus, auf das Suchen einer kurzen Rundtour durch die 100 größten Städte Deutschlands an und vergleichen Sie Ergebnisse mit denen anderer Heuristiken (etwa der Nearest-Neighbor-Heuristik, der Farthest-Insertion-Heuristik oder der Tourverschmelzung). Halten Sie hierbei – um eine gute Vergleichbarkeit zu gewährleisten – die Berechnungszeiten möglichst gleich lang.

Lösung

Folgende Funtktion implementiert die primitive Match-Funktion mittels einer Listenkomprehension.

```
1 def matchLK(M,T):
2   return [ i for i in range(len(T)) if all( T[i+j]==M[j] for j in range(len(M))) ]
```

Aufgabe 10.2

Schreiben Sie eine Python-Funktion ...

(a) ... *isReflexive*(A,R), die testet, ob die als Sequenz von Paaren übergebene Relation R reflexiv ist. Der Parameter A soll hierbei die Grundmenge spezifizieren. Beispielanwendung:

```
>>> isReflexive( [1,2,3,4], [ (1,1),(1,2),(2,2),(4,2),(3,3),(4,4) ])
>>> True
```

(b) ... *isSymmetric*(A,R), die testet, ob die als Sequenz von Paaren übergebene Relation R symmetrisch ist. Der Parameter A soll hierbei die Grundmenge spezifizieren.

(c) ... *isAntiSymmetric*(A,R), die testet, ob die als Sequenz von Paaren übergebene Relation R anti-symmetrisch ist. Der Parameter A soll hierbei die Grundmenge spezifizieren.

(d) ... *isTransitive*(A,R), die testet, ob die als Sequenz von Paaren übergebene Relation R transitiv ist. Der Parameter A soll hierbei die Grundmenge spezifizieren.

Lösung

```
def isReflexive(A,R):
  return all((x,x) in R for x in A)

def isSymmetric(A,R):
  return all((y,x) in R for (x,y) in R)

def isAntiSymmetric(A,R):
  return [(x,y) for (x,y) in R if (y,x) in R and x≠y] == []

def isTransitive(A,R):
  return all((x,z) in R for (x,y) in R for (y2,z) in R if y==y2)
```

Aufgabe 10.4

(a) Erklären Sie, warum die Laufzeit von der eben vorgestellten Python-Funktion „F“ sehr ungünstig ist und schätzen Sie die Laufzeit ab.

(b) Implementieren Sie eine nicht-rekursive Funktion *fib*(n), die die Liste der ersten n Fibonacci-Zahlen berechnet. Anstatt rekursiver Aufrufe sollten die Fibonacci-Zahlen in einer Liste gespeichert werden und bei der Berechnung des nächstens Wertes auf die schon in der Liste gespeicherten Werte zurückgegriffen werden.

(c) Geben Sie unter Verwendung von *fib* einen Python-Ausdruck an, der überprüft, ob

die Formel

$$F_{n+2} = 1 + \sum_{i=0}^{n} F_n$$

für alle $n \leq 1000$ gilt.

Lösung

(a) Die beiden rekursiven Aufrufe bewirken, dass letztlich immer wieder die gleichen Operationen ausgeführt werden. Ein Aufruf von $F(5)$ beispielsweise zieht sehr viele Aufrufe von $F(1)$ nach sich.

(b) Folgendermaßen kann eine nichtrekurvie Berechnung der Fibonacci-Folge erfolgen:

```
1 def fib(n):
2     fibs = [0,1]
3     for i in range(2,n+1):
4         fibs.append(fibs[-1]+fibs[-2])
5     return fibs
```

Den „Trick“ (wenn man ihn als solchen bezeichnen will) nennt man Tabulation: die schon gespeicherten Werte speicherten man in einer Tabelle, hier: *fibs*, und greift bei Bedarf auf diese zu).

(c) Durch den Ausdruck in der zweiten Zeile kann ermittelt werden ob die Formel für alle $n \leq 1000$ gilt.

```
>>> f = fib(1000)
>>> all(sum(f[:i])+1 == f[i+1] for i in range(1000))
```

Aufgabe 10.8

(a) Wieviele Elemente hat $\mathcal{P}(M)$?

(b) Was ist der Wert von $len(pot(pot(pot([0,1]))))$?

Lösung

(a) Sei $|M|$ gleich der Anzahl an Elementen in M. Dann gilt:

$$|\mathcal{P}(M)| = 2^{|M|}$$

Erklärung: Für jedes der $|M|$ Elemente in M gibt es zwei Möglichkeiten: entweder es ist in einem bestimmten Element von $\mathcal{P}(M)$ enthalten, oder eben nicht. Für die

beiden Möglichkeiten des ersten Elementes gibt es wiederum zwei Möglichkeiten für das zweite Element, usw. Insgesamt gibt es also

$$\overbrace{2 \cdot 2 \cdot 2 \cdot \ldots \cdot 2}^{|M|\text{–mal}} = 2^{|M|}$$

mögliche Teilmengen von M.

(b) $2^{16} = 65536$

Aufgabe 10.9

Implementieren Sie die Funktion $ins(x, xs)$, die die Liste aller möglichen Einfügungen des Elements x in die Liste xs zurückliefert. Beispielanwendung:

```
>>> ins(1, [2,3,4,5])
>>> [[1,2,3,4,5], [2,1,3,4,5], [2,3,1,4,5], [2,3,4,1,5], [2,3,4,5,1]]
```

Tipp: Am einfachsten geht eine rekursive Implementierung. Es empfiehlt sich auch die Verwendung einer Listenkomprehension.

Lösung

Die Funktion *ins*, die die Liste aller möglichen Einfügungen des Elementes x in die Liste xs zurückliefert kann folgendermaßen implementiert werden:

```
1 def ins(x,xs):
2   if xs == []: return [[x]]
3   return [[x]+xs] + [ [xs[0]]+ys for ys in ins(x,xs[1:]) ]
```

Die Implementierung erfolgt durch eine typische rekursive Implementierung. Zeile 2 regelt den Rekursionsabbruch: Es gibt eine einzige mögliche Einfügung eines Elements x in eine leere Liste, nämlich $[x]$. Bei der Implementierung des Rekursionsschrittes erfolgt der rekursive Aufruf $ins(x, xs[1:])$, der auf der kürzeren Liste $xs[1:]$. Unter der Annahme, dass dieser rekursive Aufruf das „richtige“ tut, muss man sich fragen, was man dieser „kleineren“ Lösung noch hinzufügen muss, damit die „größere“ Lösung (nämlich $ins(x, xs)$ entsteht. Lösung: Dieser kleineren Lösung fehlt das erste Element $x[0]$, das in der Listenkomprehension an jede der entstehenden Liste vorne angefügt wird. Zusätzlich fehlt noch die Liste, in der x ganz vorne eingefügt wird, also $[x]+xs$.
