Prof. G. Zachmann
Dipl.-Inf. D. Mainzer

TU Clausthal
Institut für Informatik

Klausur zu "Informatik II"

Sommersemester 2010

V/X/MMX, $14^{00} - 15^{00}$

Name:	.....	**Matrikelnummer:**	.....
Vorname:	.....		
Studiengang:	.....		
Semesterzahl:	.....		
E-Mail:	.....		

Bitte in Druckschrift ausfüllen.

Hinweise: (GENAU DURCHLESEN!)

- Neben Papier und Schreibutensilien sind keine weiteren Hilfsmittel erlaubt. Verwenden Sie keine roten Stifte und keine Bleistifte.
- Vergessen Sie nicht, Ihren Namen und die Matrikelnummer auf *jedes* Blatt zu schreiben. Blätter ohne diese Angaben werden nicht gewertet.
- Schreiben Sie Ihre Lösungen auf die Aufgabenblätter möglichst in die dafür vorgesehenen Felder. Sie können auch die Rückseiten verwenden. Weiteres Schreibpapier kann von den Betreuern angefordert werden. Benutzen Sie kein mitgebrachtes Papier.
- Bei Multiple-Choice-Fragen gibt es für jedes richtig gesetzte Kreuz einen Punkt. Für jedes falsch gesetzte Kreuz wird ein halber Punkt abgezogen! Nicht beantwortete Fragen werden nicht gewertet. Die Gesamtpunktzahl beträgt mindestens 0.
- Bitte schreiben Sie in Ihrem eigenen Interesse deutlich. Für nicht lesbare Lösungen können wir keine Punkte vergeben.
- Klausurblätter dürfen nicht voneinander getrennt werden.
- Werden mehrere unterschiedliche Lösungen für eine Aufgabe abgegeben, so wird die Aufgabe nicht gewertet.
- Im Fall von Täuschungsversuchen wird die Klausur sofort mit 0 Punkten bewertet. Eine Vorwarnung erfolgt *nicht*.

Aufgabe	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	$\sum$
max. Punkte	4	7	7	7	6	10	4	8	1	54
erreicht										

Aufgabe		4 Punkte
	Name:	(4 x 1)
1	Matrikel-Nr.:	Σ: .....

Datenstrukturen

Geben Sie jeweils eine Datenstruktur an, die geeignet ist, um folgende Aufgaben schnell zu lösen.

a) Sie sollen ein „invertiertes Telefonbuch“ speichern, d.h., mit Ihrer Datenstruktur soll es effizient möglich sein, zu gegebener Telefonnummer den Namen zu finden. Achtung: Ihr Speicher ist knapp, aber die Telefonnummern bestehen aus vielen Ziffern und sind nicht „dicht“ im Wertebereich! (D.h., ein simples Array ist keine Lösung.)

Lösung:
Hash-Tabelle oder auch Patricia-Trie

b) In Firmen wird meist derjenige Angestellte zuerst entlassen, der als letzter eingestellt wurde, wenn die Zeiten schlechter werden. (Ob das ein gutes Kriterium ist, sei hier dahingestellt.) In welcher Datenstruktur sollten die Angestelltenakten gespeichert werden, um die Reihenfolge der Entlassungen schnell bestimmen zu können?

Lösung:
Stack

c) Sie wollen eine Simulation einer stark frequentierten Kreuzung implementieren. Welche Datenstruktur verwenden Sie, um die ankommenden Autos zu speichern?

Lösung:
Queue

d) Welche Datenstruktur verwenden Sie, um ein Wörterbuch für einen Spell-Checker platzsparend zu speichern und Wörter effizient nachzuschlagen? Gesucht ist eine Datenstruktur, die nur so viel Platz benötigt, wie auch tatsächlich belegt wird, d.h., eine Hash-Tabelle kommt nicht in Betracht. (Ein “Dictionary” ist der abstrakte Datentyp; gefragt ist hier eine konkrete Datenstruktur.)

Lösung:
Binärer Suchbaum, Trie, zur Not Hash-Tabelle

Aufgabe		7 Punkte
2	Name:	(4+2+1)
	Matrikel-Nr.:	Σ: .....

Algorithmen-Design

Gesucht ist ein effizienter Algorithmus zum Potenzieren natürlicher Zahlen. Gegeben seien natürliche Zahlen x und n. Gesucht wird ein Algorithmus, der mit Aufwand $O(\log n)$ die n-te Potenz von x, d.h., die Zahl x^n berechnet. Dabei sind Addition, Subtraktion und Multiplikation zweier natürlicher Zahlen als Elementarschritte (Operationen mit konstanter Zeit) erlaubt. Gleiches gilt für das Abfragen, ob eine natürliche Zahl gerade oder ungerade ist.

a) Geben Sie einen Algorithmus in Pseudocode oder Python an, der in der geforderten Laufzeit das gewünschte Ergebnis produziert.

Lösung:

```
def quick_pow( x, n ):
  if n == 0:                                        # Base case = 1 Punkt
    return 1
  if n % 2 == 0:          # n ist gerade          # Fallunterscheidung = 1 Punkt
    x2 = quick_pow( x, n/2 )                      # Fall 1 = 1 Punkt
    return x2*x2
  else:                   # n ist ungerade          # Fall 2 = 1 Punkt
    x2 = quick_pow( x, (n-1)/2 )
    return x2*x2*x
```

b) Geben Sie die Laufzeit des Algorithmus' als Rekursionsgleichung an.

Lösung:
$T(n) = a + T(\frac{n}{2})$ (1 Punkt für 'n/2', 1 Punkt für 'a+')

c) Wie heißt die Algorithmentechnik, die Ihrem Algorithmus zugrunde liegt?

Lösung:
Divide-and-Conquer

Aufgabe		7 Punkte
3	Name:	(2 + 1 + 4)
	Matrikel-Nr.:	Σ: .....

Algorithmentechnik

Ein Freelancer-Programmierer bekommt aufgrund der guten Wirtschaftslage eine Menge von n Projekten $(P_1, \ldots, P_n)$ von n Kunden $(K_1, \ldots, K_n)$. Jedes Projekt dauert $p_i = d_i + \frac{1}{3}d_i$ Tage zur Fertigstellung. $\frac{1}{3}d_i$ ist der Zeitaufwand für die Einarbeitung und Kundengespräche, der Rest (d_i) ist für die eigentliche Programmierarbeit. Der Freelancer kann nur ein Projekt gleichzeitig bearbeiten.

Um die Kundenzufriedenheit zu maximieren, soll die Gesamtwartezeit aller Kunden minimiert werden. Die Wartezeit eines Kunden ist die benötigte Zeit zur Fertigstellung der vorher bearbeiteten Projekte plus die Zeit zur Fertigstellung seines Projektes.

a) Geben Sie einen Algorithmus an, der die Gesamtwartezeit minimiert.

Lösung:
Sortiere P nach p aufsteigend.
Arbeite von vorne nach hinten diese Liste ab.

b) Welche Algorithmentechnik verwendet Ihr Algorithmus?

Lösung:
Greedy-Algorithmus. (Dynamisches Programmieren geht auch, ist aber völlig Overkill.)

c) Zeigen Sie, dass Ihr Algorithmus tatsächlich das Minimum berechnet. (Sie dürfen hier den Stoff der Vorlesung als bekannt annehmen, d.h., Sie dürfen z.B. sagen "...; ab hier weiter wie im Optimalitätsnachweis zu Algorithmus X".)

Lösung:

$$
\begin{aligned}
\text{Sei } I =& (i_1 i_2 \ldots i_n) \text{ eine Permutation} \\
T(I) =& p_{i_1} + (p_{i_1} + p_{i_2}) + (p_{i_1} + p_{i_2} + p_{i_3}) + \ldots \\
=& n p_{i_1} + (n-1) p_{i_2} + \ldots \\
=& \sum_{k=1}^{n} (n-k+1) p_{i_k} \\
=& \frac{3}{4} \sum_{k=1}^{n} (n-k+1) d_{i_k}
\end{aligned}
$$

Ab hier weiter wie im Beweis zum Greedy-Algorithmus für das Scheduling-Problem.

Aufgabe		7 Punkte
4	Name:	(7)
	Matrikel-Nr.:	Σ: .....

Sortieralgorithmen

Erläutern Sie die wesentlichen Unterschiede zwischen den Sortieralgorithmen Quicksort und Mergesort. Welches sind ihre Gemeinsamkeiten? (. . . außer, dass beides Sortieralgorithmen sind!) Gefragt sind hier Charakteristika, nicht die Unterschiede im Ablauf der Algorithmen.

Lösung:

- Quicksort: in situ; Mergesort: nicht in situ (in place) (1P)
- Worst-Case von Quicksort: $O(n^2)$; Worst-Case von Mergesort: $O(n \log n)$ (2P)
- Mergesort eignet sich als externes Sortierverfahren (“unbegrenzte” Datengröße), Quicksort nicht. (1P)
- Gemeinsamkeit: Beide sind Divide-and-Conquer (1P)
- Gemeinsamkeit: beide sind $O(n \log n)$ im average case. (2P)

Aufgabe		6 Punkte
5	Name:	(4+1+1)
	Matrikel-Nr.:	Σ: .....

Typsysteme

a) Geben Sie an, was von folgenden Python-Code-Zeilen ausgegeben wird:

```
a = 1
b = "zwei"
d = [a, b]
a = 3.0
print a
print b
print d
```

Tip: denken Sie daran, dass Python *dynamic binding* verwendet.

Lösung:

```
>>> print a   # 1P
3.0
>>> print b   # 1P
zwei
>>> print d   # 2P
[1, 'zwei']
```

b) Betrachten Sie folgende Funktion in Python:

```
def swap( a, b ):
  t = a
  a = b
  b = t
```

Dürfen `a` und `b` beliebigen Typ haben? ☑ Ja ☐ Nein

c) Markieren Sie diejenige Programmzeile, die in einer stark typisierten (*strongly typed*) Sprache (wie z.B. Python) zu einer Fehlermeldung führt.

```
a = 2
b = "drei"
a = "zwei"
b = 3.1
c = a + b
```

Lösung:
letzte Zeile

Aufgabe		10 Punkte
6	Name:	(10)
	Matrikel-Nr.:	Σ: .....

Doppelt verkettete Listen

Gegeben seien folgende Klassen in Python zur Implementierung von doppelt verketteten Listen.

```
class List:
  class ListElement:
    def __init__( self ):
      self.item = self.next = self.prev = None
  def __init__( self ):
    self.cursor = None  # cursor always points to some "current" element,
                        # except when the list is empty
```

Implementieren Sie in dieser Klasse eine Methode *insert_second*(*self*, *item*), die ein Datenelement `item` (z.B. einen Integer) immer als *zweites* Element in die Liste einhängt.

Tip: evtl. geht es am einfachsten, wenn Sie zunächst den Python-Code für den einfachen Fall auf der Rückseite vorschreiben, dann die Behandlung der Spezialfälle einfügen, und dann das Ganze auf der Vorderseite ins Reine schreiben. (Durchstreichen des vorgeschriebenen Konzeptes nicht vergessen!)

Lösung:

```
def insert_second( self, item ):                    # 1P
  if self.cursor == None:          # empty list     # 1P
    return
  old_cursor = self.cursor           # optional     # 1 Extra-P
  while self.cursor.prev != None:    # find head     # 1P
    self.cursor = self.cursor.prev
  # now cursor = head
  if self.cursor.next == None:                      # 1P
    # no 2nd element in list ... easy case
    self.cursor.next = self.ListElement()           # 1P
    self.cursor.next.prev = self.cursor             # 1P
  else: # a little bit more to do ...
    old_2nd = self.cursor.next
    new_2nd = self.ListElement()
    old_2nd.prev = new_2nd
    new_2nd.prev = self.cursor      # 1P
    new_2nd.next = old_2nd          # 1P
    self.cursor.next = new_2nd      # 1P
  self.cursor.next.item = item      # 1P
  self.cursor = old_cursor           # optional
```

Aufgabe		4 Punkte
7	Name:	(4)
	Matrikel-Nr.:	Σ: .....

Komplexitätsanalyse

Betrachten Sie folgende Funktion zum Sortieren:

```
1 def ssort( A ):
2     for i in range( 0, len(A) ):
3         min = i
4         for j in range( i+1, len(A) ):
5             if A[j] < A[min]:
6                 min = j
7         A[i], A[min] = A[min], A[i]  # swap
```

Dabei ist `A` ein Array von Integers.

Leiten Sie die Komplexität $T(n)$ der Funktion her, wobei n die Anzahl der Elemente in `A` bezeichnet. Sie dürfen das Groß-O-Kalkül benutzen.

Lösung:
Innere Schleife: $O(n-i)$ Äußere Schleife: $T(n) \in \sum_{i=0}^{n} O(n-i) = O(\sum_{i=0}^{n}(n-i)) = O(\sum_{j=0}^{n} j)$[mit $j = n-i$] $= O(\frac{n(n+1)}{2}) = O(n^2)$

Aufgabe		8 Punkte
8	Name:	(8 x 1)
	Matrikel-Nr.:	Σ: .....

Quickies

Kreuzen Sie für jede Aussage an, ob sie wahr oder falsch ist.

a) Der Aufwand der Suche nach einem Element in einer doppelt verketteten Liste beträgt im worst-case $O(\log n)$. ☐ Richtig ☑ Falsch

b) Für Integer-Keys ohne beschränkten Wert gibt es ein Sortierverfahren, dessen Komplexität im Worst-Case in $O(n)$ ist. ☐ Richtig ☑ Falsch

c) Es gibt Sortierverfahren, deren Komplexität im Worst-Case in $O(\log n)$ ist. ☐ Richtig ☑ Falsch

d) Ein binärer Baum mit h Levels hat $O(2^h)$ viele Knoten. ☑ Richtig ☐ Falsch

e) In einer Hash-Tabelle mit *Open Addressing* zur Kollisionsbehandlung kann man mehr Elemente speichern als die Tabelle Slots hat. ☐ Richtig ☑ Falsch

f) Einen optimalen binären Suchbaum kann man in der Zeit $O(n^3)$ konstruieren. ☑ Richtig ☐ Falsch

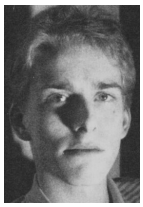
g) In einem beliebigen Baum kann man für zwei beliebige Knoten v und w in konstanter Zeit entscheiden, ob v Vorgänger von w ist. ☑ Richtig ☐ Falsch

h) In einem B-Baum mit Knotenbelegungsparameter k haben alle Blätter dieselbe Höhe. ☑ Richtig ☐ Falsch

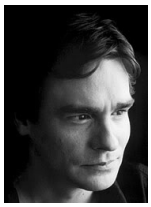
Aufgabe		1 Punkt
9	Name:	(1)
	Matrikel-Nr.:	Σ: .....

Informatik II

Welche der hier abgebildeten Personen ist der Dozent der Vorlesung “Informatik II” im Sommersemester 2010?


☐


☐


☐


☐


☑


☐


☐