Klausuraufgaben

Algorithmen und Datenstrukturen

14. Juli 2009

Zur Lösung der nachfolgenden Aufgaben haben Sie **90 Minuten** Zeit. Es sind keine Hilfsmittel zugelassen!

- Für alle Aufgaben ist der **Lösungsweg** und die Lösung **nachvollziehbar** anzugeben, d.h. Sie müssen mich von Ihrer Lösung mittels dem Lösungsweg überzeugen! Antworten ohne Begründung, d.h. ohne Lösungsweg, oder zu schlechte Handschrift, d.h. nicht lesbarer Schrift, werden nicht bewertet!
- Bitte schreiben Sie zuerst Ihren Namen und Ihre Matrikelnummer zumindest auf das erste Blatt und auf alle weiteren (losen) Blätter zumindest die Matrikelnummer!

Die Aufgaben sind nicht nach dem Schwierigkeitsgrad geordnet!

Aufgabe	1	2	3	4	5	$\sum$
	Grundlagen	Komplexität	Sortieren	Bäume	Hashing	Summe
mögliche Punkte	22	28	31	19	14	114
erreichte Punkte						

Erreichte Bewertung:

Aufgabe 1 (Grundlagen) *(22P)*

1. **Algorithmus** *(5 Punkte)*

   (a) *Beschreiben Sie den Begriff Algorithmus und erklären Sie die vier allgemeinen Anforderungen an einen solchen Algorithmus.*

   (b) *Erklären Sie folgende vier Eigenschaften eines Algorithmus:* terminiert, determiniert, deterministisch und nicht deterministisch.

2. **Datenabstraktion** *(8 Punkte)*

   (a) *(1P) Beschreiben Sie das Prinzip bzw. die Grundidee der Datenabstraktion.*

   (b) *(7P) Definieren Sie durch Angabe der formalen Beschreibung einen Datentyp* `Stack`*, der einen Stapel nach dem Last-In-/First-Out-Prinzip (LiFo) modelliert. Geben Sie dazu die Wertebereiche (Typen) und die Operationen an. Bei den Operationen ist die allgemeine/formale Typisierung anzugeben (z.B. durch Kreuzprodukt) sowie die Pre- und Postcondition.*

   Definieren Sie Operationen zum Ablegen eines Elementes (`push`*), zum Entfernen des ersten Elementes (*`pop`*), zum Abfragen des ersten Elementes (*`top`*) sowie zur Bestimmung der Höhe des Stack (*`high`*).*

3. **Rekursion** *(9 Punkte) Gegeben sei nachfolgende Funktion:*

$$f(n) = \begin{cases} 1 & \text{falls } \forall n \in \mathbb{Z} : n \leq 1 \\ f(n-1) + f(n-2) & \text{falls } \forall n \in \mathbb{Z} : n > 1 \end{cases}$$

   (a) *Implementieren Sie diese Funktion direkt, also ohne Optimierungsüberlegungen,in Java.*

   (b) *Wie groß ist die Anzahl der Additionen $A(n)$ die beim Aufruf von $f(n)$ ausgeführt werden ? Berechnen Sie dazu zuerst die Anzahl der Additionen für alle $n \in \{1, 2, 3, 4, 5, 6, 7\}$. Versuchen Sie dann eine allgemeine Formel aufzustellen.*

   (c) *Wie kann man die Funktion f* **rekursiv** *so implementieren, dass sie im Zeitaufwand linear läuft, also $O(n)$ besitzt ? Transformieren Sie dazu ggf. Laufzeitaufwand der naiven Lösung in Speicherplatzaufwand, jedoch mit maximal $O(n)$ Speicherplatzaufwand! Geben Sie dazu ein Java-Programm an und begründen kurz, warum der Aufwand jeweils (Laufzeit-/Speicherplatz) linear ist.*

Aufgabe 2 (Komplexitätstheorie) *(28P)*

1. **Aufwandabschätzung** *(6 Punkte)*
   Gegeben sei folgendes Codefragment; n sei eine positive natürliche Zahl.

```
int j = 0, i = n;
while (i > 0) {
 for(j = i; j <= n; j++) {
  AnweisungY; }
 for(j = n; j > 0 ; j--) {
  AnweisungY; }
 AnweisungY;
 i--; }
```

   Geben Sie eine Funktion f an, die in Abhängigkeit von n bestimmt, wie oft AnweisungY ausgeführt wird. Ordnen Sie den Algorithmus der bestmöglichen (niedrigsten) Komplexitätsklasse O in Abhängigkeit von n zu.

2. **Break-even Punkt** *(5 Punkte) Für die Aufgabe, n Adressen zu sortieren, stehen drei Verfahren zur Verfügung: Eine Implementierung von Heapsort benötigt eine Zeit von* $240 * log_2(n)\mu s$ *, eine Implementierung von Bubblesort genau* $3 * n^2 \mu s$ *und eine Implementierung von Insertion Sort* $2 * n^2 + 6n + 16\ \mu s$*. Für welche Datenbankgrössen n lohnt sich welches der Sortierverfahren? Bestimmen Sie die „Break-even"-Punkte! Für den Break-even-Punkt mit Heapsort kann kurz vor der formalen Auflösung, d.h. einer Gleichung ohne* log_2*, mit den beiden Werten* $n = 22$ *und* $n = 21$ *getestet werden.*

3. **Aufwandabschätzung** *(16 Punkte)*
Gegeben sei folgendes Codefragment; n sei eine positive natürliche Zahl.

```
01 public static double foo(double a[], int i, int j) {
02 int mid;
03 double foo1, foo2 ;
04 if ( i == j ) return a[ i ] ;
05 else {
06    mid = (int) (((double) (i + j) ) / 2.0) ;
07    foo1 = foo( a, i, mid ) ;
08    foo2 = foo( a, mid+1, j ) ;
09    if ( foo1 > foo2 ) return foo1 ;
10   else return foo2 ; }
}
```

(a) *Welchen Wert liefert der Aufruf* `foo(a,1,8)` *zurück, wenn die Elemente* `a[1], ... , a[8]`, *mit den Werten* `3.5, 5.5, 12.5, 4.5, 6.5, 2.5, 0.5, 7.5` *belegt sind ? Begründen Sie Ihre Antwort durch Angabe des Ablaufes!*

(b) *Was berechnet der Aufruf* `foo( a, 1, n )` *im Allgemeinen ? (Die Feldelemente a[1], ..., a[n] seien mit reellen Zahlen vorbelegt.). Begründen Sie Ihre Antwort durch allgemeine Erklärungen an Hand des Algorithmus (dazu ggf. die Zeilennummern verwenden)!*

(c) *Auf welchem algorithmischen Konstruktionsprinzip basiert die Funktion* `foo` *? Denken Sie auch hier an die Begründung Ihrer Antwort!*

(d) *Geben Sie eine möglichst kleine O-Schranke für die Zeitkomplexität des Aufrufs* `foo( a, 1, n )` *(in Abhängigkeit von n) an. Zur Vereinfachung dürfen Sie annehmen, dass n von der Form* $n = 2^k, k \in \mathbb{N}_0$ *ist.*

Sie können den Aufwand einer Rückgabe, Division und eines Vergleiches jeweils mit 1 berechnen.

Tip*:*

$$\sum_{i=1}^{n} i = \frac{n * (n+1)}{2}$$

Aufgabe 3 (Sortieren) *(31P)*

1. **Einfaches Verfahren** *(11 Punkte)*
   Implementieren Sie (inklusive der Funktion `swap`*) ein einfaches Sortierverfahren für integer-Arrays, das nach folgendem Grundprinzip funktioniert:*

```
Solange gilt: Es gibt einen Index i
in dem Array a mit a[i] > a[i+1] tue
             Führe swap(i, i+1) aus.
```

   Erklären Sie Ihre Umsetzung in konkretem Bezug zu diesem Grundprinzip. Bei der Implementierung darf nur ein Schleifenkonstrukt verwendet werden!

   Sortieren Sie mit Ihrer Implementierung folgendes array: $a = \{32, 3, 86, 0\}$. Geben Sie nach jedem Durchgang Ihrer Schleife den Zustand des aktuellen arrays an.

2. **Shell Sort** *(6 Punkte)*
   Sotieren Sie die folgende Zahlenreihe mit dem im Tip aufgeführten Shell Sort Algorithmus (nicht mit dem ggf. in Ihrem Gedächtnis vorhandenem Algorithmus!). Notieren Sie die Zwischenergebnisse nach jeder Einfügeoperation (, d.h. den Zustand nach Ausführung von Zeile `13`*) sowie die jeweilige Distanz h.*

a[0]	*a[1]*	*a[2]*	*a[3]*	*a[4]*	*a[5]*	*a[6]*	*a[7]*	*a[8]*	*a[9]*
46	*45*	*21*	*6*	*119*	*99*	*12*	*118*	*74*	*41*

   Geben Sie die insgesamte Anzahl der vorgenommenen Vertauschungen (Ausführung von Zeile `11`*) an.*

3. **Heapsort** *(8 Punkte)*
   Gegeben sei ein Maximum-Heap, wie in der Vorlesung vorgestellt. Beantworten Sie folgende Fragen und begründen Sie Ihre Antworten:

   (a) *Was versteht man hierbei unter dem Begriff „Heap-Eigenschaft“?*

   (b) *Wie wird die Baumrepräsentation eines Heaps in ein Array kodiert, also wo findet man die Nachfolger eines Knotens im Array?*

   (c) *Was ist die minimale und maximale Anzahl von Elementen in einem Heap der Höhe h?*

   (d) *Welche Elemente in einer unsortierenden Zahlenfolge bilden zu Anfang bereits einen Heap, und warum wird diese Struktur von hinten (rechts) nach vorne (links) in der Array-Repräsentation eines Heaps aufgebaut?*

4. **Heapsort** *(6 Punkte)*
Gegeben sei folgender Maximum-Heap:
`( 45,31,41,23,15,12,32,10,8,2,13 )`.

   (a) *Zeichnen Sie den Heap als Baum.*

   (b) *Fügen Sie nacheinander die Elemente 28 und 60 ein. Der Vorgang ist so zu dokumentieren, dass der Einfügeprozess nachvollzogen werden kann. Geben Sie dazu z.B. die vorgenommenen Vertauschungen an.*

   (c) *Löschen Sie das Wurzelelement des nach dem Einfügen entstandenen Heaps, und stellen Sie die „Heap-Eigenschaft" wieder her. Der Vorgang ist so zu dokumentieren, dass der Einfügeprozess nachvollzogen werden kann. Geben Sie dazu z.B. die vorgenommenen Vertauschungen an.*

Tip*:*

- **Algorithmus Shell Sort**

```
01   void ShellSort(int N) {
02      int h = 1, i = 0;
03      /* Anfangswert für die Distanz nach Knuth */
04      for ( h = 1; h <= (N/9); h = 3*h+1);
05      for( ; h>0; h /= 3) {
06         for(i = h; i <= N ; i=i+h) {
07            int j = i;
08            int t = a[i];
09            int k = t.key;
10            while(((j-h)>= 0) && (a[j-h] > t)) {
11               a[j] = a[j-h];
12               j = j-h; }
13            a[j] = t; }
14      } }
```

- **Algorithmus Heapsort**

```
int HeapEnde = N;
HeapSeep (int i,int HEnde) {
   while ((2*i) <= HEnde) {
      int j = 2*i; int kl = a[2*i].key; int kr = kl-1;
      if ((2*i+1) <= HEnde) { int kr = a[2*i+1].key; };
      if (kl < kr) j = j+1;
      if (a[i].key < a[j].key) { swap(i,j); i = j;
       } else { i = HEnde + 1; };
   } }
reHeap_up( ) { // Heapeigenschaft herstellen
   for (int i = HeapEnde/2; i > 0; i--) HeapSeep(i,HeapEnde); }
reHeap_down( ) { // komplette Sortierung!
   for (int i = HeapEnde; i > 2; i--) {
      swap(1,i); HeapSeep(1,i); } }
insert(int elem) {
   HeapEnde++;
   a[HeapEnde] = elem;
   int vater = HeapEnde/2; int sohn = HeapEnde;
   while (vater > 0) {
      if (a[vater] < a[sohn]) {
         swap(vater,sohn);
         sohn = vater; vater = sohn/2;
      } else { vater = 0; } } }
int pop( ) {
   int elem = a[1];
   a[1] = a[HeapEnde];
   HeapEnde--;
   HeapSeep(1,HeapEnde);
   return elem; }
```

Aufgabe 4 (Bäume) *(19P)*

1. **AVL-Baum** *(15 Punkte)*
   Nachfolgende Implementierung für den AVL-Baum muß fertig gestellt werden. Folgender, für die Implementierung benötigter Code liegt dazu vor:

```
public class AVLnode {
  int key = -1, hoehe = -1, counter = 0;
  AVLnode links = null, rechts = null;

  public AVLnode (int wert){
     key = wert; hoehe = 0; counter = 1;
     links = rechts = null; } }


public class AVLtree {
   private static AVLnode wurzel = null;
   ...
   public static void insert(int c) {
      wurzel = insert(wurzel, c); }
   public static AVLnode insert(AVLnode n, int insert) {
      /* TODO */ }

   public static AVLnode rotiererechts(AVLnode n) {
      /* TODO: Rechtsrotation UND Hoehen neu bestimmen!  */ }

   public static AVLnode rotierelinks(AVLnode n) {
      /* TODO: Linksrotation UND Hoehen neu bestimmen!  */ }

   public static int hoehe(AVLnode n) {
      if (n == null) return -1;
      else return n.hoehe; }
   public static AVLnode pruefen(AVLnode n) {
      /* FERTIG: prueft Balance und rotiert ggf. für Knoten n */
      ... } }
```

   Implementieren Sie rekursiv den fehlenden Code für die Funktion `insert`*, die einen Wert in den AVL-Baum einfügt und auf eine Rebalancierung prüft (linker Teilbaum ist für kleinere Werte zuständig). Bei Mehrfachvorkommen eines Wertes ist ein Zähler in dem zugehörigen Knoten zu inkrementieren. Implementieren Sie zudem die beiden Rotationen* `rotierelinks` *für die Links- bzw.* `rotiererechts` *für die Rechtsrotation (Siehe Abbildung 1, Seite 9).*

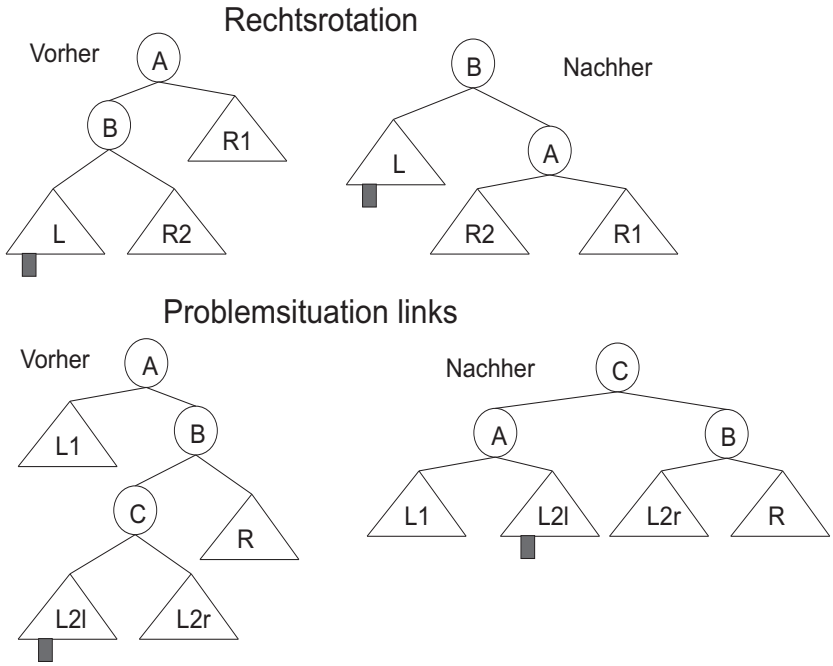


Abbildung 1: Rotationsarten eines AVL-Baumes

2. **AVL-Baum** *(8 Punkte)*
Fügen Sie mit dem in der vorangegangenen Aufgabe von Ihnen implementierten Algorithmus folgende Zahlen **in der vorgegebenen Reihenfolge** *in einen leeren AVL-Baum ein:* $(4, 5, 8, 6, 9, 7, 1, 0, 2, 3)$. *Geben Sie die Anzahl der durchgeführten Rechts- und Linksrotationen an. Geben Sie am Ende den Baum in Inorder als eine Zeile aus.*

Sofern Sie die vorangegangene Aufgabe (noch) nicht gelöst haben, geben Sie bitte in Pseudo-Code an, nach welcher Methode Sie in den AVL-Baum Elemente einfügen.

Tip: **AVL-Bedingungen** *für Rotationen (Siehe Abbildung 1 (Seite 9)):*

1. **Rechtsrotation**: *Die Höhe des Teilbaums* `R1` *ist um 2 niedriger, als die Höhe des Teilbaums mit Wurzel* `B`*. Der Unterschied sei durch den Teilbaum* `L` *begründet. Auf dem Teilbaum mit Wurzel* `A` *wird eine Rechtsrotation durchgeführt.*

2. **Linksrotation**: *analog der Rechtsrotation*

3. **Problemsituation links**: *(Doppelte Linksrotation) Die Höhe des Teilbaums* `L1` *ist um 2 niedriger, als die Höhe des Teilbaums mit Wurzel* `B`*. Der Unterschied sei durch den Teilbaum* `L2` *begründet. Auf dem Teilbaum mit Wurzel* `B` *wird eine Rechtsrotation durchgeführt und dann wird auf dem Baum mit Wurzel* `A` *eine Linksrotation durchgeführt.*

4. **Problemsituation rechts**: *(Doopelte Rechtsrotation) analog der Problemsituation links.*

Aufgabe 5 (Hashing) *(14P)*

1. **Konstruktionsprinzip** *(6 Punkte)*

   (a) *Suchalgorithmen, die mit Hashing arbeiten, bestehen aus zwei Teilen. Welche sind dies?*

   (b) *Welche Ziele verfolgen die einzelnen Teile?*

   (c) *Warum ist Hashing ein gutes Beispiel für einen Kompromiss zwischen Zeit- und Platzbedarf?*

2. **Mittel-Quadrat-Methode** *(8 Punkte)*
   Fügen Sie die Zahlen

   $$13, 6, 27, 11, 16, 17, 19, 18, 42$$

   in dieser Reihenfolge unter Verwendung folgender Hashfunktion in eine Hashtabelle der Größe $N = 11$ *(nummeriert von 0 bis 10) ein:*

   $$qk = k^2; \quad st = qk[3, 2]; \quad h(st) = st \; mod \; 11$$

   Dabei liefert $qk[3, 2]$ *die Stellen 3 und 2 (von rechts zählend), also z.B.* $5291763[3, 2] = 76$ *oder* $54321[3, 2] = 32$ *sowie* $12[3, 2] = 1$. *Die Kollisionsbehandlung erfolge durch* $h_i(k) = (h_0(k) + i^2) \; mod \; 11$.

   Das Einfügen ist so zu dokumentieren, dass der Ablauf nachvollziehbar ist. Geben Sie z.B. hierzu auch jeweils das Ergebnis der Hashfunktion an.

Die nachfolgenden Lösungen verstehen sich als Musterlösungen, d.h. auch andere Lösungen bzw. Lösungswege sind möglich.

Lösung 1 (Grundlagen) *(22P)*

1. **Algorithmus** *(5 Punkte)*

   (a) *(1P) Für die Informatik hat der Algorithmus die Bedeutung einer eindeutigen Vorschrift zur Lösung eines Problems mit Hilfe eines Computers.*

   Zu nennen waren zudem die vier Anforderungen an ein Verfahren, das den Namen „Algorithmus“ verdient: (2P)

   Eindeutigkeit *: Jeder elementare Schritt ist unmißverständlich beschrieben und läßt keine Wahlmöglichkeit offen.*

   Endlichkeit *: Die Beschreibung des Algorithmus hat eine endliche Länge (durch Schleifen o.Ä. kann das ablaufende Programm allerdings durchaus unendlich lange brauchen, wenn der Algorithmus fehlerhaft ist).*

   Terminiertheit *: Ein aktuelles Programm, das diesen Algorithmus implementiert, soll in endlicher Zeit beendet sein.*

   Reproduzierbarkeit *: Der Algorithmus soll unabhängig von einer speziellen Implementation sein. Verschiedene (korrekte) Implementationen sollen dasselbe Ergebnis liefern,*

   (b) *(2P) Folgende Eigenschaften kann man u.U. beobachten:*

   - *Liefert der Algoritmus ein Resultat nach endlich vielen Schritten, so sagt man, der Algorithmus* **terminiert**.
   - *Ein Algorithmus heißt* **determiniert**, *falls er bei gleichen Eingaben und Startbedingungen stets dasselbe Ergebnis liefert.*
   - *Ein Algorithmus heißt* **deterministisch**, *wenn zu jedem Zeitpunkt seiner Ausführung höchstens eine Möglichkeit der Fortsetzung besteht.*
   - *Ein Algorithmus heißt* **nicht deterministisch**, *wenn mindestens zu einem Zeitpunkt seiner Ausführung eine Wahlmöglichkeit bei der Fortsetzung besteht.*

2. **Datenabstraktion** *(8 Punkte)*

(a) *(1P) Mit der Datenabstraktion stehen Methoden zur Verfügung, mit denen die Art der Verwendung eines Datenobjektes von den Details seiner Konstruktion aus elementareren Datenobjekten getrennt werden kann, d.h. dass nicht bekannt ist, wie die verwendete Datenstruktur „intern“ realisiert ist.*

Die Grundidee bei der Datenabstraktion besteht darin, die Programme zur Verwendung von zusammengesetzten Datenobjekten so zu strukturieren, dass sie mit „abstrakten Daten“ arbeiten. Gleichzeitig wird eine „konkrete“ Darstellung der Daten unabhängig von den die Daten verwendenden Programmen definiert.

(b) *Hier die formale Beschreibung des ADT (2P):*

ADT Stack			
push:	$stack \times elem$	$\rightarrow$	*stack*
pop:	*stack*	$\rightarrow$	*stack*
top:	*stack*	$\rightarrow$	*elem*
high:	*stack*	$\rightarrow$	*integer*

Dabei sind stack der ADT `stack`*, elem die auf dem* `stack` *ablegbaren Elemente und integer die Menge der ganzen Zahlen (1P).*

Hier nun die Pre- und Postconditions der einzelnen Operationen (4P):

Operation push:	$stack \times elem \rightarrow stack$; $s = push(s, b)$
pre:	*keine*
post:	*ist* $s = (a_1, \ldots, a_n)$, *so bewirkt* $s = push(s, b)$, *dass* $s = (a_1, \ldots, a_n, b)$
	ist $s = (\quad)$ *also leer, so bewirkt* $s = push(s, b)$, *dass* $s = (b)$
Operation pop:	$stack \rightarrow stack$; $s = pop(s)$
pre:	*keine*
post:	*ist* $s = (a_1, \ldots, a_n)$, *so bewirkt* $s = pop(s)$, *dass* $s = (a_1, \ldots, a_{n-1})$
	ist $s = (\quad)$ *also leer, so bewirkt* $s = pop(s)$, *dass* $s = (\quad)$.
Operation top:	$stack \rightarrow elem$; $e = top(s)$
pre:	*keine*
post:	*ist* $s = (a_1, \ldots, a_n)$, *so bewirkt* $e = top(s)$, *dass* $e = a_n$
	ist $s = (\quad)$ *also leer, so bewirkt* $e = top(s)$, *dass* $e =$ *null als Fehlercode*
Operation high:	$stack \rightarrow integer$; $h = high(s)$
pre:	*keine*
post:	*ist* $s = (a_1, \ldots, a_n)$, *so bewirkt* $h = high(s)$, *dass* $e = n$
	ist $s = (\quad)$ *also leer, so bewirkt* $h = high(s)$, *dass* $e = 0$

3. **Rekursion** *(9 Punkte) Gegeben war nachfolgende Funktion, die man normalerweise als* Fibonacci-Zahlen *bezeichnet:*

$$f(n) = \begin{cases} 1 & \textit{falls } \forall x \in \mathbb{Z} : x \leq 1 \\ f(n-1) + f(n-2) & \textit{falls } \forall x \in \mathbb{Z} : x > 1 \end{cases}$$

(a) *Hier eine Implementierung in Java (3P):*

```
...
public class rekursion {
...

public static int f(int n) {
   int ergebnis = 0;
   if (n <= 1) { ergebnis = 1;
   } else { ergebnis = f(n-1) + f(n-2); }
   return ergebnis;}

public static void main(String args[]) {
   int myn = 0;
   if (args.length > 0) myn = Integer.parseInt(args[0],10);
   int dummy = f(myn);
}
```

(b) *Hier die Anzahl der Additionen für alle* $n \in \{1, 2, 3, 4, 5, 6, 7\}$ *(2P):*

n	1	2	3	4	5	6	7
$f(n)$	1	2	3	5	8	13	21
$A(n)$	0	1	2	4	7	12	20

Die allgemeine Formel wäre: $A(n) = f(n) - 1 = A(n-1) + A(n-2) + 1$.

(c) Hier eine Implementierung in Java (als Erweiterung des vorangegangenen Programms) (2P):

```
...
   private static int[ ] speicher;
...
public static int fs(int n) {
   int ergebnis = 0;
   if (n <= 1) { ergebnis = 1;
   } else {
       ergebnis = fs(n-1) + speicher[n-2];
   }
   speicher[n] = ergebnis;
   return ergebnis; }

public static void main(String args[]) {
...
   speicher = new int[myn+1];
   speicher[0] = 1;
   dummy = fs(myn);
}
```

(2P) Die Idee besteht darin, den zweiten Rekursionsaufruf durch die Verwendung eines Gedächtnisses, also Speicherplatzes, zu ersetzen, womit eine lineare Rekursion und damit ein linearer Aufwand entsteht (der Aufwand pro Rekursionsaufruf ist durch die Addition konstant). Der Speicherplatz ist offensichtlich als array linear abhängig von n.

Lösung 2 (Komplexitätstheorie) *(28P)*

1. **Aufwandabschätzung** *(6 Punkte)*
   Die äussere `while`*-Schleife wird insgesamt n-mal durchgeführt (1P). Die erste innere* `for`*-Schleife wird insgesamt*

$$1 + 2 + \ldots + n = \sum_{i=1}^{n} i = \frac{n * (n+1)}{2}$$

   mal durchgeführt (1,5P). Die zweite innere `for`*-Schleife wird bei jedem Durchgang der äusseren* `while`*-Schleife n-mal durchgeführt und damit insgesamt* $n * n = n^2$ *mal durchgeführt (1P). Wir erhalten somit für die AnweisungY insgesamt (1,5P):*

$$f(n) = \frac{n * (n+1)}{2} + n^2 + n = 3 * \frac{n * (n+1)}{2}$$

   Ausführungen. Dabei ergibt sich die Summe aus Anzahl der ersten `for`*-Schleife plus Anzahl der zweiten* `for`*-Schleife plus Anzahl der* `while`*-Schleife.*

   Nehmen wir für die AnweisungY einen konstanten Aufwand an, so ergibt sich für die Funktion wegen $3 * \frac{n*(n+1)}{2} = \frac{3}{2}n^2 + \frac{3}{2}n$ *einen Aufwand von* $O(n^2)$ *(1P).*

2. **Break-even Punkt** *(5 Punkte) Die Zeiten werden in Relation zueinander gestellt: Zunächst Bubblesort und Insertion-Sort, wobei wir annehmen, dass Insertion-Sort ab einer gewissen Größe von n schneller ist, als Bubblesort (2,5P):*

Bubblesort langsamer-als Insertion-Sort $\leftrightarrow$

$$3*n^2 > 2*n^2+6n+16 \quad \leftrightarrow_{-2*n^2-6n} \quad n^2-6n > 16 \quad \leftrightarrow \quad (n-3)^2-9 > 16 \quad \leftrightarrow_{+9}$$

$$(n-3)^2 > 25 \quad \leftrightarrow_{\sqrt[2]{x};\, n>0} \quad n-3 > 5 \quad \leftrightarrow_{+3} \quad n > 8$$

   Der Break-even-Punkt liegt bei $n = 8$*, d.h. ab* $n = 8$ *ist die Implementierung von Insertion-Sort schneller als die Implementierung von Bubblesort (0,5P).*

   Nun vergleichen wir Heapsort mit Insertion-Sort, wobei wir annehmen, dass Heapsort ab einer gewissen Größe von n schneller ist, als Insertion-Sort (1,5P):

Heapsort schneller-als Insertion-Sort $\leftrightarrow$

$$240*log_2(n) < 2*n^2+6n+16 \quad \leftrightarrow_{*\frac{1}{240}} \quad log_2(n) < \frac{n^2}{120}+\frac{n}{40}+\frac{1}{15} \quad \leftrightarrow_{2^x} \quad n < 2^{\frac{n^2+3n+8}{120}}$$

   Wir berechnen nun für $n = 22$ *und* $n = 21$ *die Werte:*

$$22 < 2^{\frac{558}{120}} \quad \leftrightarrow \quad 22 < 25,1067 \qquad 21 \not< 2^{\frac{512}{120}} \quad \leftrightarrow \quad 21 \not< 19,2484$$

   Der Break-even-Punkt liegt bei $n = 21$*, d.h. für alle* $n > 21$ *ist die Implementierung von Heapsort schneller als die Implementierung von Insertion-Sort (0,5P), da dann* $n < 2^{\frac{n^2+3n+8}{120}}$ *gilt.*

3. **Aufwandabschätzung** *(16 Punkte)*
Hier die einzelnen Lösungen:

(a) *Die Ablauf ist wie folgt (3P):*
Rekursionsteilung mit (1,4) und (5,8) führt zum Ergebnis 12.5.
Rekursionsteilung mit (1,2) und (3,4) führt zum Ergebnis 12.5.
Rekursionsteilung mit (1,1) und (2,2) führt zum Ergebnis 5.5.
Rekursionsende bei 1 erreicht führt zum Ergebnis 3.5.
Rekursionsende bei 2 erreicht führt zum Ergebnis 5.5.
Rekursionsteilung mit (3,3) und (4,4) führt zum Ergebnis 12.5.
Rekursionsende bei 3 erreicht führt zum Ergebnis 12.5.
Rekursionsende bei 4 erreicht führt zum Ergebnis 4.5.
Rekursionsteilung mit (5,6) und (7,8) führt zum Ergebnis 7.5.
Rekursionsteilung mit (5,5) und (6,6) führt zum Ergebnis 6.5.
Rekursionsende bei 5 erreicht führt zum Ergebnis 6.5.
Rekursionsende bei 6 erreicht führt zum Ergebnis 2.5.
Rekursionsteilung mit (7,7) und (8,8) führt zum Ergebnis 7.5.
Rekursionsende bei 7 erreicht führt zum Ergebnis 0.5.
Rekursionsende bei 8 erreicht führt zum Ergebnis 7.5.

Das Ergebnis hier ist daher das Maximum aller Zahlen: 12.5.

(b) *(1P) Zeile* `04` *stellt sicher, dass die Aufteilung bei einem Feldelement aufhört und gibt dieses als Wert zurück.*
(1,5P)Zeile `06` *stellt durch Bildung des arithmetischen Mittels sicher, dass irgendwann die Intervallsgröße auf i gerundet wird, und mit* `mid+1` *in Zeile* `08` *auch auf j „gerundet“ wird.*
Somit wird sichergestellt, dass alle Felder irgendwann als Rückgabewert eines (rekursiven) Aufrufes von `foo` *auftreten.*

(1P) Zeile `09` *stellt sicher, dass als Resultat von den beiden rekursiven Aufrufen das größere Element, als das Maximum der beiden Zahlen zurück gegeben wird.*

(0,5P) Da also alle Felder irgendwann als Rückgabewert auftreten und da jeweils das Maximum zweier Rückgabewerte zurück gegeben wird, berechnet die Funktion insgesamt das Maximum aller eingegebenen Zahlen.

(c) *(1P) Die Aufteilung erfolgt hier ähnlich zu dem Mergesort-Algorithmus und folgt damit auch dem Konstruktionsprinzip des Teile-und-Herrsche Prinzips. Die Maximumsbildung von n-Zahlen wird dadurch (geteilt) reduziert auf die Maximumsbildung (beherrscht) von zwei Zahlen.*

(d) *(8P)Für die O-Schranke benötigt man eine rekurrente Gleichung, die wir C nennen:*

(1P) Für den Rekursionsabbruch (Zeile `04`*) gilt:* $C(1) = 1$*, d.h. für die Rückgabe wird ein konstanter Aufwand von* 1 *angenommen.*

(2P) Im Falle der Rekursion wird durch die Bildung des arithmetischen Mittel eine gleichmässige Aufteilung erreicht und man erhält: $C(n) = 1 + C(\frac{n}{2}) + C(\frac{n}{2}) + 1$*, mit der Annahme, dass die Division (Zeile* `06`*) wie auch der Vergleich (Zeilen* `09-10`*) einen konstanten Aufwand benötigen. Es ergibt sich somit* $C(n) = 2 + 2 * C(\frac{n}{2})$*. Unter der Vorraussetzung, dass* $n = 2^k, k \in \mathbb{N}_0$ *ergibt sich (1P):*

$$C(2^k) = 2 + 2 * C(\frac{2^k}{2}) = 2 + 2 * C(2^{k-1})$$

Also z.B. (1,5P)

$$C(2^1) = 2 + 2 * C(2^0) = 2^1 + 2^1 = 2^2$$

$$C(2^2) = 2 + 2 * C(2^1) = 2^1 + 2 * (2^1 + 2^1) = 2^1 + 2^3$$
$$C(2^3) = 2 + 2 * C(2^2) = 2^1 + 2 * (2^1 + 2^3) = 2^1 + 2^2 + 2^4$$
$$C(2^4) = 2 + 2 * (2^1 + 2^2 + 2^4) = 2^1 + 2^2 + 2^3 + 2^5$$

Damit ist offensichtlich, dass für $n > 1$ *gilt (1,5P):*

$$C(n) = (\sum_{i=1}^{log_2(n)+1} 2^i) - 2^{log_2(n)}$$

(Dies kann per Induktion über n *nachgewiesen werden oder für eine Klausur reicht auch ein nachrrechnen für* $n = 2^4$*).*

Für die O-Komplexität ist der höchste Koeffizient wichtig und wir erhalten damit (1P): $O(2^{log_2(n)+1})$*. Für* $n = 2^k, k \in \mathbb{N}_0$ *wäre dies also* $O(2^{k+1}) = O(n)$*, also ein linearer Aufwand. Damit haben wir jedoch gegenüber einer naiven Methode nichts gewonnen.*

Lösung 3 (Sortieren) *(31P)*

1. **Einfaches Verfahren** *(11 Punkte)*
   Hier der Code in Java:

```
01 void swap(int i,int j) {
02        int tmp = a[i];
03        a[i] = a[j];
04        a[j] = tmp; }

05 void bubbles ( ) {
06        boolean NICHTfertig = true;
07        int i = 0;
08        while (NICHTfertig) {
09          if (a[i] > a[i+1]) {
10             swap(i,i+1);
11             i = 0;
12          } else i++;
13          if (i >= a.length - 1) NICHTfertig = false;
14        } }
```

   (1P) In den Zeilen 01 *bis* 04 *ist die Funktion* swap *implementiert. Wie üblich wird hier unter Verwendung einer temporären Variablen eine Vertauschung vorgenommen.*

   (4P) In den nachfolgenden Zeilen 05 *bis* 14 *ist das Sortierverfahren implementiert.*

   Erklärung (3P): In Zeile 08 *ist als einziges Schleifenkonstrukt die* Solange gilt *Anweisung direkt umgesetzt worden.* NICHTfertig *kodiert dabei den Zustand, dass gilt* Es gibt einen Index i in dem Array a mit a[i] $>$ a[i+1]. *In den Zeilen* 11 *und* 13 *wird diese Variable berechnet. Die Vorgehensweise ist wie folgt: wird der Zustand* a[i] $>$ a[i+1] *erkannt, wird getauscht und dann wieder von vorne begonnen (Zeile* 11*). Erreicht der Index i die Größe* $a.length - 1$*, ist dieser Fall nicht aufgetreten und die Schleife kann beendet werden.*

   Die Sortierung ergibt (links nach recht lesend, Zeilenweise) (3P):

```
(32, 3, 86, 0) i = 0        (3, 32, 86, 0) i = 0
(3, 32, 86, 0) i = 1        (3, 32, 86, 0) i = 2
(3, 32, 0, 86) i = 0        (3, 32, 0, 86) i = 1
(3, 0, 32, 86) i = 0        (0, 3, 32, 86) i = 0
(0, 3, 32, 86) i = 1        (0, 3, 32, 86) i = 2
```

2. **Shell Sort** *(6 Punkte)*
Hier der Ablauf (entspricht einer `print`*-Anweisung nach Zeile* `13`*) (5P):*

46	*45*	*21*	*6*	*119*	*99*	*12*	*118*	*74*	*41*
a[0]	*a[1]*	*a[2]*	*a[3]*	*a[4]*	*a[5]*	*a[6]*	*a[7]*	*a[8]*	*a[9]*
h = 4	*2P*								
46	*45*	*21*	*6*	*/119/*	*99*	*12*	*118*	*74*	*41*
46	*45*	*21*	*6*	*74*	*99*	*12*	*118*	*/119/*	*41*
h = 1	*3P*								
45	*46/*	*21*	*6*	*74*	*99*	*12*	*118*	*119*	*41*
21	*45*	*46/*	*6*	*74*	*99*	*12*	*118*	*119*	*41*
6	*21*	*45*	*46/*	*74*	*99*	*12*	*118*	*119*	*41*
6	*21*	*45*	*46*	*74/*	*99*	*12*	*118*	*119*	*41*
6	*21*	*45*	*46*	*74*	*99/*	*12*	*118*	*119*	*41*
6	*12*	*21*	*45*	*46*	*74*	*99/*	*118*	*119*	*41*
6	*12*	*21*	*45*	*46*	*74*	*99*	*118/*	*119*	*41*
6	*12*	*21*	*45*	*46*	*74*	*99*	*118*	*119/*	*41*
6	*12*	*21*	*41*	*45*	*46*	*74*	*99*	*118*	*119/*

(1P) Die Anzahl der Vertauschungen ist 18.

3. **Heapsort** *(8 Punkte)*
Gegeben sei ein Maximum-Heap, wie in der Vorlesung vorgestellt.

   (a) *(1P) Die „Heap-Eigenschaft“ ist wie folgt durch die Definition eines Heap beschrieben: Eine Folge* $F = k_1, k_2, \ldots, k_N$ *von Schlüsseln nennen wir einen* Heap *wenn* $k_i \leq k_{\lfloor \frac{i}{2} \rfloor}$ *für* $2 \leq i \leq N$ *gilt. Gleichbedeutend damit ist* $k_i \geq k_{2i}$ *und* $k_i \geq k_{2i+1}$*, falls* $2i, 2i+1 \leq N$*.*

   (b) *(1P) Die Baumrepräsentation eines Heaps wird wie folgt in ein Array kodiert: Die Nachfolger von Knoten i stehen an den Positionen* $2i$ *(linker Nachfolger) und Position* $2i+1$ *(rechter Nachfolger). Entsprechend ist der Vorgänger von Knoten i an der Position* $\lfloor \frac{i}{2} \rfloor$ *zu finden. Dies gilt, insofern Nachfolger oder Vorgänger tatsächlich vorhanden sind.*

   (c) *(3P) Ein Heap ist als Array kontinuierlich aufgebaut, d.h. es gibt keine Lücken im Array und damit auch nicht im binären Baum: In einem Heap der Höhe h ist also in der untersten Ebene der Baumdarstellung zumindest ein Knoten vorhanden. Maximal ist in einem Heap der Höhe h die unterste Ebene komplett aufgefüllt, d.h. sie besitzt* 2^h *Knoten (1,5P).*

Unter der Verwendung, dass ein vollständiger binärer Baum der Höhe $h-1$ 2^h-1 Knoten beinhaltet, ergibt sich (1,5P):

minimal *:* $(2^h-1)+1=2^h$ *Knoten.*

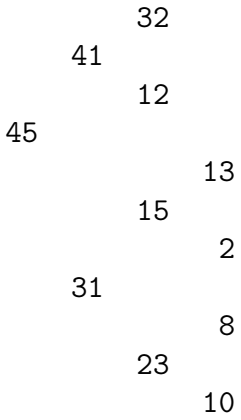
maximal *:* $(2^h-1)+2^h=2^{h+1}-1$ *Knoten.*

(d) (3P) Die Blätter in einer unsortierenden Zahlenfolge bilden zu Anfang bereits einen Heap, da sie keine Nachfolger haben und somit die „Heap-Eigenschaft" nicht verletzen können. (1P)

Die Struktur wird von hinten (rechts) nach vorne (links) in der Array-Repräsentation eines Heaps bzw. von unten nach oben in der Baumrepräsentation eines Heaps als Heap aufgebaut, da die grösseren Elemente so gesichert nach links bzw. nach oben wandern können. Denn man kennt in dieser Bottom-Up Vorgehensweise alle bisher im Heap enthaltenen Elemente, was notwendig ist, um zu entscheiden, welches das größte Element ist. Wenn man von links nach rechts wandern würde, also von oben nach unten, kennt man noch nicht das wirklich größte Element (2P).

4. **Heapsort** *(6 Punkte)*

(a) (1P) Der Baum wird der Einfachheit halber von links nach rechts gemalt:



(b) Hier wird der Heap der Einfachheit halber wieder als Array gezeichnet. Zum Einfügen wird das neue Element an das Heapende geschrieben. Entlang der Vorgänger wird nun das Element solange steigen gelassen, bis der Vorgänger größer als das neue Element ist.

Wir erhalten nach dem Einfügen von 28 (28=a[12] wurde mit 12=a[6] getauscht) (1P):
(45, 31, 41, 23, 15, 28, 32, 10, 8, 2, 13, 12)

Wir erhalten nach dem Einfügen von 60 (60=a[13] wurde mit 28=a[6], 41=a[3] und 45=a[1] getauscht) (2P):
(60, 31, 45, 23, 15, 41, 32, 10, 8, 2, 13, 12, 28)

(c) *Hier wird der Heap der Einfachheit halber wieder als Array gezeichnet. Zum Löschen wird das letzte Element an die erste Position geschrieben. Das Heapende wird dann um 1 kleiner. Nun wird ab Position 1 das Element solange versunken gelassen, bis beide Nachfolger kleiner als das kopierte Element sind.*

Wir erhalten nach dem Löschen der 60(28=a[1] wurde mit 45=a[3] und 41=a[6] getauscht) (2P):
(45, 31, 41, 23, 15, 28, 32, 10, 8, 2, 13, 12)

Lösung 4 (Bäume) *(19P)*

1. **AVL-Baum** *(15 Punkte)*
   Nachfolgend der erstellte Code (7P für `insert`*, (6+2=)8P für* `rotiere*`*),*
   :

```
public static void insert(int c) {
   wurzel = insert(wurzel, c); }

public static AVLnode insert(AVLnode n, int insert) {
   if (n == null) return new AVLnode(insert);
   if (insert < n.key) n.links = insert(n.links, insert);
   if (insert > n.key) n.rechts = insert(n.rechts, insert);
   if (insert == n.key) {
      n.counter++;
      return n; }
   n = pruefen(n);
   return n; }

public static AVLnode rotiererechts(AVLnode n) {
   AVLnode linkes = n.links;
   n.links = n.links.rechts;
   linkes.rechts = n;
   n.hoehe = Math.max(hoehe(n.rechts),hoehe(n.links)) + 1;
   linkes.hoehe = Math.max(hoehe(linkes.rechts),
                           hoehe(linkes.links)) + 1;
   return linkes; }

public static AVLnode rotierelinks(AVLnode n) {
   AVLnode rechtes = n.rechts;
   n.rechts = n.rechts.links;
   rechtes.links = n;
   n.hoehe = Math.max(hoehe(n.rechts),hoehe(n.links)) + 1;
   rechtes.hoehe = Math.max(hoehe(rechtes.rechts),
                            hoehe(rechtes.links)) + 1;
   return rechtes; }

... }
```

2. **AVL-Bäume** *(8 Punkte)*
Die Baäume sind der Einfachheit halber von links nach rechts, statt von oben nach unten dargestellt: Einfügen: $4, 5, 8$*, Linksrotation um* 4 *(1P):*

```
      8
5
      4
```

Einfügen: $6, 9, 7$*, Rechtsrotation um* 8 *und Linksrotation um* 5 *(2P):*

```
            9
      8
            7
6
      5
            4
```

Einfügen: 1*, Rechtsrotation um* 5 *(1P):*

```
            9
      8
            7
6
            5
      4
            1
```

Einfügen: $0, 2, 3$*, Linksrotation um* 1 *und Rechtsrotation um* 4 *(2P):*

```
            9
      8
            7
6
                  5
            4
                  3
      2
            1
                  0
```

(2P) Die Anzahl der Links- und Rechtsrotationen entspricht jeweils 3. In Inorder ausgegeben erhält man: $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$.

Lösung 5 (Hashing) *(14P)*

1. **Konstruktionsprinzip** *(6 Punkte)*

   (a) *Suchalgorithmen, die mit Hashing arbeiten, bestehen aus folgenden zwei Teilen:*

      i. *(1P) Im ersten Schritt transformiert der Algorithmus den Suchschlüssel mithilfe einer Hashfunktion in eine Tabellen-/Speicheradresse. Diese Funktion bildet im Idealfall unterschiedliche Schlüssel auf unterschiedliche Adressen ab. Oftmals können aber auch zwei oder mehrere unterschiedliche Schlüssel zur gleichen Tabellenadresse führen.*

      ii. *(1P) Somit führt eine Hashing-Suche im zweiten Schritt eine Kollisionsbeseitigung durch, die sich mit derartigen Schlüsseln befasst.*

   (b) *Das Ziel der einzelnen Teile ist:*

      i. *(1P) Die Hashfunktion versucht eine eventuell vorhandene Häufung der Schlüssel,d.h. z.B. eine aufsteigende Nummerierung, aufzuheben, also das Ziel ist eine gleichmässige Verteilung der Schlüssel auf alle Tabellen-/Speicherplätze zu erreichen, ohne von der Verteilung der Schlüssel selbst beeinflusst zu werden.*

      ii. *(1P) Für die Kollisionsbeseitigung gibt es zwei Möglichkeiten: Zum Einen kann auf ein anderes Verfahren zurückgegriffen werden, z.B. lineare Listen oder ausgeglichene Bäume. Zum Anderen kann rekursiv gearbeitet werden: Hierbei ist das Ziel, durch die Kollisionsbeseitigung möglichst wenige Kollisionen für die nachfolgenden Einfügevorgänge zu erzeugen.*

   (c) *(2P) Hashing ist ein gutes Beispiel für einen Kompromiss zwischen Zeit- und Platzbedarf. Wenn es keine Speicherbeschränkung gäbe, könnte jede Suche mit nur einem einzigen Speicherzugriff realisiert werden, indem einfach der Schlüssel als Speicheradresse analog zu einer schlüsselindizierten Suche verwendt wird. Allerdings läßt sich dieser Idealzustand meistens nicht verwirklichen, weil der Speicherbedarf besonders bei langen Schlüsseln viel zu groß ist. Gäbe es andererseits keine Zeitbeschränkung, könnten wir mit einem Minimum an Speicher auskommen, indem ein sequenzielles Suchverfahren eingesetzt wird. Hashing erlaubt es, zwischen diesen beiden Extremen ein vertretbares Maß sowohl für die Zeit als auch den Speicherplatz zu finden. Insbesondere können wir jedes beliebige Verhältnis einstellen, indem wir lediglich die Größe der Hashtabelle anpassen; dazu brauchen wir weder Code neu zu schreiben noch auf andere Algorithmen auszuweichen.*

2. **Mittel-Quadrat-Methode** *(8 Punkte)*
Die Kollisionsbehandlung wird in eigenen Zeilen beschrieben.

a[0]	*a[1]*	*a[2]*	*a[3]*	*a[4]*	*a[5]*	*a[6]*	*a[7]*	*a[8]*	*a[9]*	*a[10]*
-	-	-	-	-	-	-	-	-	-	-
$(13^2)[3,2]mod11 = 5$										
-	-	-	-	-	*13*	-	-	-	-	-
$(6^2)[3,2]mod11 = 3$										
-	-	-	*6*	-	*13*	-	-	-	-	-
$(27^2)[3,2]mod11 = 6$										
-	-	-	*6*	-	*13*	*27*	-	-	-	-
$(11^2)[3,2]mod11 = 1$										
-	*11*	-	*6*	-	*13*	*27*	-	-	-	-
$(16^2)[3,2]mod11 = 3$										
$(3+1^2)mod11 = 4$										
-	*11*	-	*6*	*16*	*13*	*27*	-	-	-	-
$(17^2)[3,2]mod11 = 6$										
$(6+1^2)mod11 = 7$										
-	*11*	-	*6*	*16*	*13*	*27*	*17*	-	-	-
$(19^2)[3,2]mod11 = 3$										
$(3+1^2)mod11 = 4$										
$(3+2^2)mod11 = 7$										
$(3+3^2)mod11 = 1$										
$(3+4^2)mod11 = 8$										
-	*11*	-	*6*	*16*	*13*	*27*	*17*	*19*	-	-
$(18^2)[3,2]mod11 = 10$										
-	*11*	-	*6*	*16*	*13*	*27*	*17*	*19*	-	*18*
$(42^2)[3,2]mod11 = 10$										
$(10+1^2)mod11 = 0$										
42	*11*	-	*6*	*16*	*13*	*27*	*17*	*19*	-	*18*