Max Wisniewski , Alexander Steen

Tutor: Lena Schlipf

Aufgabe 1

Die Funktionen der Aufgabe sollen derart geordnet werden, so dass $g_i \in \Omega(g_{i+1})$ gilt. Geben Sie auch an, wenn sogar $g_i = \Theta(g_{i+1})$ ist.

Die Folge erfüllt die Eigenschaft und enthält die Elemente, die geordent werden müssen:

$$(g_i)_{1\leq i\leq 10} = (2^{(2^n)}, 2^n, n^{log(log(n))}, (\lceil \log n \rceil)!, 4^{logn}, n^2, (\sqrt{2})^{\log n}, \log^2 n, \ln n, n^{\frac{1}{\log n}})$$

Was wir an dieser Stelle benutzen werden, sind die Definitionen:

1. $f \in O(g) \Leftrightarrow 0 \leq \limsup_{\lim x\to\infty} \left|\frac{f(x)}{g(x)}\right| < \infty$
2. $f \in \Omega(g) \Leftrightarrow \exists c > 0\ \exists x_0 > 0\ \forall x > x_0\ :\ |f(x)| \geq c \cdot |g(x)|$
3. $f \in \Theta(g) \Leftrightarrow 0 < \lim_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| < \infty$

Und der Satz:

4. $f \in \Omega(g) \Leftrightarrow g \in O(f)$
5. $\lim_{n\to\infty} f(n) = c \Leftrightarrow \exists g, h\ :\ g > f \wedge h < f\ \wedge\ \lim_{n\to\infty} h(n) = c\ \wedge\ \lim_{n\to\infty} g(n) = c$ (Sandwich Kriterium)

Die Äquivalenz der jeweils passenden Definitionen wurde in vorangegangenen Vorlesung gezeigt, genau so wie die Richtigkeit des Satzes.

Beweis

1. $2^n \in O(2^{(2^n)})$:
$\lim_{n\to\infty} \frac{2^n}{2^{(2^n)}} = \lim_{n\to\infty} \frac{2^n \cdot 1}{2^n \cdot 2^{(2^n)-n}} = \lim_{n\to\infty} \frac{1}{2^{(2^n)-n}}$
Da 2^n stärker wächst als n, gilt:
$\lim_{n\to\infty} \frac{1}{2^{(2^n)-n}} = 0$
Damit gilt nach konvergenz Kriterium $2^n \in \Omega(2^{(2^n)}) \Rightarrow g_1 \in \Omega(g_2)$

2. $n^{\log(\log(n))} \in O(2^n)$:
Es gilt: $n^{\log\log n} = 2^{\log n \cdot \log\log n}$

$$\lim_{n\to\infty} \frac{2^{\log n \cdot \log\log n}}{2^n} = \lim_{n\to\infty} 2^{(\log n \cdot \log\log n) - n} = 0 \Leftarrow \lim_{n\to\infty} \log(n + \log n) - n = -\infty$$

Nun können wir das asymptotische Verhalten der beiden Terme betrachten, um festzustellen, ob sie gegen $-\infty$ fallen, d.h. gegen 0, wenn man den vorderen Term durch den hinteren Teilt:

$$\lim_{\lim x\to\infty} \frac{\log n \cdot \log\log n}{n} \overset{\text{alles} > 0}{\leq} \lim_{\lim x\to\infty} \frac{2 \cdot \log n}{n} \overset{\text{l'Hôpital}}{=} \lim_{\lim x\to\infty} \frac{1}{n} = 0$$

Aus unserer Vorüberlegung folgt: $n^{\log(\log n)} \in O(2^n) \Rightarrow g_2 \in \Omega(g_3)$

3. $(\lceil \log n \rceil)! \in O(n^{log(\log(n))})$:

Beweisskizze: Wir werden zeigen, dass die Funktion $\frac{(\lceil \log n \rceil)!}{n^{log(\log(n))}}$ innerhalb von Zweierpotenzen streng monoton fällt und danach, dass die Eckpunkte, das heißt die Zweierpotenzen an sich, gegen 0 konvergieren. Damit kann man schlussfolgern, das eine Kurve, die durch die Punkte an den Zweierpotenzen geht, wie der lim sup konvergiert, da die Funktion niemals größer als diese springen wird. Als erstes ist zu beachten, dass $\frac{(\lceil \log n \rceil)!}{n^{log(\log(n))}} > 0$ gilt, da keiner dieser Terme bei $n \geq 0$ eine negative Zahl produziert.

Eckpunkte konvergieren: Die Folge $t_m = \frac{(\lceil \log 2^m \rceil)!}{n^{log(\log(2^m))}} = \frac{m!}{m^m}$ konvergiert gegen 0.
Beweis:

$$\begin{aligned} \lim_{m\to\infty} t_m = \lim_{m\to\infty} \frac{m!}{m^m} &= \lim_{m\to\infty} \prod_{i=1}^{m} \frac{i}{m} \\ &= \left(\lim_{m\to\infty} \frac{1}{m}\right) \cdot \left(\lim_{m\to\infty} \prod_{i=2}^{m} \frac{i}{m}\right) \end{aligned}$$

Nun können wir den hinteren Term abschätzen, da $\forall 1 < i \leq m \ : \ 0 \leq \frac{i}{m} \leq 1$ gilt. Damit können wir den gesammte Produkt, gegen eine Zahl $0 \leq k \leq 1$ aschätzen.

$$\begin{aligned} \Rightarrow \lim_{m\to\infty} t_m &= \lim_{m\to\infty} \frac{1}{m} \cdot k \\ &= 0 \end{aligned}$$

Wir wissen nun, dass die Eckpunkte gegen 0 konvergieren.

Intervalle fallen: Auf dem Interval $(n, 2n)$, für n der Form $n = 2^m$ fällt die Funktion.
Zu beachten, zwischen $\log 2^m = m$ und $\log 2^{m+1} = m + 1$ liegt keine natürliche Zahl. Deshalb kann auf dem Interval eingeschänkt dieser Teil als Konstante angenommen werden.

$\Rightarrow \frac{k}{n^{\log(\log n)}}$ die Funktion im Nenner wächst, da jede Einzelkomponente wächst.

Wir haben nun gezeigt, dass die zwischen 2 Zweierpotenzen die Funktion fällt, und das die Folge der Anfangspunkte gegen 0 konvergiert.
Damit gilt : $\lim_{n\to\infty} t_m = \limsup_{n\to\infty} \frac{g_3(n)}{g_4(n)} = 0$. Damit gilt nach Definition $g_4 \in O(g_3)$
$\Rightarrow g_3 \in \Omega(g_4)$

4. $4^{\log(n)} \in O((\lceil \log n \rceil)!)$:
Wie wir im nächsten Schritt zeigen gilt: $4^{\log n} = n^2$.
Wir bedienen uns hier des gleichen Kriteriums, wie im letzen Schritt.

Punkte fallen: für $n = 2^m$, nach $2n = 2^{m+1}$z.z.

$$\frac{2^{2m+2}}{\lceil \log 2^{m+1} \rceil!} = \frac{4^{m+1}}{(m+1)!} = \frac{4}{m+1} \cdot \frac{4^m}{m!} \overset{\text{für } m>3}{<} \frac{4^m}{m!} = \frac{n^2}{\lceil \log n \rceil!}$$

Intervallmaxima fallen monoton Da wir diesmal wieder eine konstante annehmen können, sieht die Folge so aus: $\frac{1}{k} \cdot n^2$. Da $n > 0$, wird n^2 streng

monoton steigen. Deshalb und aus dem ersten Punkt folgt, dass das Maximum kurz vor der letzen Zweierpotenz liegen muss, also kurz bevor der Nenner einen neuen Wert annimmt.z.z.:

$$\frac{(2^{m+1}-1)^2}{(m)!} < \frac{(2^m-1)^2}{(m-1)!}$$

Beweis:

$$\begin{aligned}\frac{(2^{m+1}-1)^2}{(m)!} &= \frac{(2\cdot 2^m-1)^2}{m\cdot(m-1)!}\\ &= \frac{4\cdot 2^{2m}-4\cdot 2^m+1}{m\cdot(m-1)!}\\ &= \frac{3\cdot 2^{2m}-2\cdot 2^m}{m}\cdot\frac{(2^m-1)^2}{(m-1)!}\\ &< \frac{(2^m-1)^2}{(m-1)!}\end{aligned}$$

Nun lässt sich zeigen (was in dieser Stelle ausgenommen wurde), dass beide Folgen (obere und untere) gegen 0 fallen. Nach Sandwich Kriterium muss also auch die eingeschlossene Folge gegen 0 konvergieren.
$\Rightarrow g_4 \in \Omega(g_5)$

5. $n^2 \in \Theta(4^{\log(n)})$:
Wir zeigen an dieser Stelle, das gilt: $n^2 = 4^{\log(n)}$. Damit gilt die Beziehung für Θ sofort.
$4^{\log n} = (2^2)^{\log n} = 2^{2\cdot\log n} = 2^{\log n^2} = n^2$
$\Rightarrow g_5 \in \Theta(g_6)$

6. $(\sqrt{2})^{\log n} \in O(n^2)$:
Ersteinmal gilt : $(\sqrt{2})^{\log n} = (2^{\frac{1}{2}})^{\log n} = 2^{\frac{1}{2}\cdot\log n} = 2^{\log\sqrt{n}} = \sqrt{n}$
Nun wenden wir wieder das Konvergenzkreterium an:
$\lim_{n\to\infty}\frac{\sqrt{2}}{n^2} = \lim_{n\to\infty}\frac{1}{n^{1.5}} = 0$
$\Rightarrow (\sqrt{2})^{\log n} \in O(n^2) \Rightarrow g_6 \in \Omega(g_7)$

7. $\log^2 n \in O(\sqrt{n})$:

$$\lim_{n\to\infty}\frac{\log^2 n}{\sqrt{n}} \overset{\text{l'Hôpital}}{=} \lim_{n\to\infty}\frac{4\cdot\sqrt{n}\cdot\log n}{n} = \lim_{n\to\infty}\frac{4\cdot\log n}{\sqrt{n}}$$

$$\overset{\text{l'Hôpital}}{=} \lim_{n\to\infty}\frac{8\cdot\sqrt{n}}{n} \overset{\text{l'Hôpital}}{=} \lim_{n\to\infty}\frac{4}{\sqrt{n}} = 0$$

An dieser Stelle sollte offensichtlich sein, dass es gegen 0 konvergiert.
$\Rightarrow g_7 \in \Omega(g_8)$

8. $\ln n \in \Omega(\log^2 n)$:

$$\lim_{n\to\infty}\frac{\ln n}{\log^2 n} = \lim_{n\to\infty}\frac{(\ln 2)(\log n)}{\log^2 n} = \lim_{n\to\infty}\frac{\ln 2}{\log n} = 0$$

Nach Konvergenzkriterium gilt: $g_8 \in \Omega(g_9)$

9. $n^{\frac{1}{\log n}} \in \Omega(\ln n)$:
Zunächst gilt: $n^{\frac{1}{\log n}} = 2^{\frac{\log n}{\log n}} = 2$.
Daraus folgt offensichtlich:
$\lim_{n\to\infty}\frac{2}{\ln n} = 0 \Rightarrow g_9 \in \Omega(g_{10})$

Aufgabe 2

Sei n die Anzahl der verschiedenen Sammelbilder. Sei X Zufallsvariable für die Anzahl der benötigten Packungen Müsli, bis wir alle n Sammelbilder haben. Die Wahrscheinlichkeit für ein bestimmtes Bild ist gleichverteilt.

(a)

$E[X]$ soll die Anzahl der Runden sein, bis man alle Bilder hat. $E[X_i]$ soll beschreiben, wie viele Runden man in Runde i braucht, das heißt wie viele Runden wir von $i-1$ bis i Bilder brauchen. Nun brauchen wir alle Bilderkarten $n = \sum_{i=1}^{n} i$.Die Zufallsvariable X_i beschreibt die Anzahl der Runden, das heißt für alle Runden brauchen wir $X = \sum_{i=1}^{n} X_i$.
Nach der Linearität des Erwartungswertes gilt:

$$E[X] = E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i]$$

(b)

Als erstes zeigen wir, dass wir für $E[X_i]$ eine geometrische Verteilung ist, d.h. das gilt: $Pr(X_i = k) = (1-p_i)^{k-1} \cdot p_i$

Gehen wir unseren Wahrscheinlichkeitsbaum hinunter haben wir jedesmal die möglichkeit, dass wir eine Karte bekommen, die wir schon haben oder wir bekommen eine neue. Eine neue Karte erhalten wir mit p_i, dass heißt, wenn wir keine Karte bekommen, haben wir jedesmal den Zweig mit den $1-p_i$ genommen, der Gegenwahrscheinlichkeit. Erhalten wir nun in der $k$ten Runde eine neue Karte, so müssen wir in den $k-1$ Runden vorher keine erhalten haben. $\Rightarrow Pr(X_i = k) = (1-p_i)^{k-1} \cdot p_i$.
Wir haben für $E[X_i]$ eine geometrische Verteilung.
$E[X_i] = \frac{1}{p_i}$, mit p_i Wahrscheinlichkeit ein neues Bild (das i-te Bild) zu erhalten.
Für p_i ergibt sich $p_i = \frac{n-i+1}{n}$, da wir in Runde 1 eine Wahrscheinlichkeit von $p_1 = \frac{n-1+1}{n} = 1$, in Runde 2 $p_2 = \frac{n-2+1}{n} = \frac{n-1}{n}$, etc. haben, ein neues Bild zu erhalten.
Dann ist

$$E[X_i] = \frac{1}{p_i} = \frac{1}{\frac{n-i+1}{n}} = \frac{n}{n-i+1}$$

(c)

z.z.: $E[X] = O(n \log n)$

$$E[X] \overset{(a)}{=} \sum_{i=1}^{n} E[X_i] \overset{(b)}{=} \sum_{i=1}^{n} \frac{n}{n-i+1} = n \sum_{i=1}^{n} \frac{1}{n-i+1}$$

$$\overset{(*)}{=} n \sum_{i=1}^{n} \frac{1}{i} = n \cdot O(\log n) = O(n \log n)$$

Aufgabe 3

(a)

Selectionsort

Beschreibung: Solange bis die zu sortierende Liste leer ist, sucht Selectionsort das kleinste Element, löscht dieses aus der Liste und fügt es hinten an eine am Anfang neu erzeugte Liste an.

Laufzeit: Um das kleinste Element in einer unsortierten Liste zu finden, muss man sich jedes Element einmal ansehen. Bei einer Liste der Länge n bedeutet dies $T_{smallest}(n) = n$.
Nach der obigen Beschreibung nehmen wir n mal das kleinste Element aus der Liste, wobei die Liste in jedem Schritt um ein Element schrumpft. Um das Element hinten an die neue Liste zu hängen, können wir konstante laufzeit annehmen. (LinkedList rear oder Array auf das ende + 1).

$$\begin{aligned} \Rightarrow T(n) &= \sum_{i=0}^{n-1} i \\ &= \frac{n\cdot(n-1)}{2} \\ &= \frac{1}{2}\left(n^2 - n\right) \end{aligned}$$

Mergesort

Beschreibung: Teile die Liste rekursiv in 2 gleichgroße Listen, bis die Teile trivial zu lösen sind, z.B. bei einem Element, und vereinige danach die sortierten Teillisten auf dem Weg den Rekursionsbaum hoch wieder, in der richtigen Reihenfolge.

Laufzeit: In jedem Schritt Teilen wir die Liste in 2 Teile. Haben wir Listen der Größe 1 erreicht, können wir aufhören. Diese sind ohne weiteres sortiert. Nach jedem Teilen benötigen wir noch n Schritte um beide Listen zu mergen (da wir im Worst-Case immer abwechselnd ein Element aus den jeweiligen Listen nehmen müssen)

$$\begin{aligned} \Rightarrow T(1) &= 0 \\ T(2) &= 1 \text{ ,aus folgender Formel ausgerechnet} \\ T(n) &= T(\lfloor \tfrac{n}{2} \rfloor) + T(\lceil \tfrac{n}{2} \rceil) + n - 1 \quad , n > 1 \end{aligned}$$

Da n eine Zweierpotenz ist, können wir $n = 2^k$ substituieren.

$$\Rightarrow T(2^k) = T\left(\left\lfloor 2^{k-1} \right\rfloor\right) + T(\left\lceil 2^{k-1} \right\rceil) + 2^k - 1$$

Nun gilt $\forall a > 0 : 2^{n-1} \in \mathbb{N}$ und da $n > 1 \Rightarrow k > 0$.

$$\Rightarrow T(2^k) = 2 \cdot T(2^{k-1}) + 2^k - 1$$

Nun stellen wir die charakteristische Gleichung für den homogenen Teil auf:

$$x^k = 2 \cdot x^{k-1} \Leftrightarrow x = 2$$

Als Lösung des inhomogenen Teils wählen wir $k \cdot 2^k$:

$$\begin{array}{rcl} T(2^k) & = & c_1 \cdot 2^k + c_2 \cdot k \cdot 2^k \\ \overset{\text{resub.}}{\Leftrightarrow} T(n) & = & c_1 \cdot n + c_2 \cdot n \cdot \log n \\ & T(1) = 0 & \\ T(1) = 0 & = & c_1 \cdot 1 + c_2 \cdot 1 \cdot \log 1 \\ 0 & = & c_1 \\ & T(2) = 1 & \\ T(2) = 1 & = & 0 \cdot 1 + c_2 \cdot 2 \cdot \log 2 \\ 1 & = & c_2 \cdot 2 \\ c_2 & = & \frac{1}{2} \end{array}$$

Setzen wir dies in unsere aufgelöste Form ein erhalten wir:

$$\begin{array}{rcl} T(n) & = & c_1 \cdot n + c_2 \cdot n \cdot \log n \\ & = & 0 \cdot n + \frac{1}{2} \cdot n \log n \\ & = & \frac{1}{2} \cdot n \cdot \log n \end{array}$$

Die konstante von $\frac{1}{2}$ macht soweit Sinn, als dass wir auf der letzten Ebene unseres Baumes keine Vergleiche mehr brauchen, sondern nur auf den Ebenen darüber.

(b)

Nachdem wir in der letzten Aufgabe die inhomogene Rekursion schon gelöst haben, können wir jetzt einfach die Gleich erneut mit anderem Anker auflösen. In der Rechnung ist zu beachten, das wir $m \leq 1$ annehmen, da schon bei dieser Größe keine echte Sortierung per Selection mehr passiert.

$$\begin{array}{rcl} T(m) = \frac{m^2-m}{2} & = & 2c_1 m + 2c_2 m \cdot \log m \\ T(2m) = m^2 + m & = & 2c_1 m + 2c_2 m \log(2m) \end{array}$$

Anker m bschreibt die Größe, wenn wir Selektionsort ausführen und die nächste 2er Potenz (die wir annehmen durften) ist bei $2m$ zu finden.

$$\begin{array}{rcl} II - I: \ 1 & = & c_2 \cdot \log(2m) - c_2 \cdot \log m \\ 1 & = & c_2 \cdot (\log(2m) - \log m) = c_2 \\ \text{in II } m^2 - m & = & 2c_1 \cdot m + 2m \log m \\ c_1 & = & \frac{m-1}{2} - \log m \end{array}$$

Nun können wir die konstanten einsetzen und die Formel endgültig aufstellen:

$$T(n) = \begin{cases} \frac{n^2-n}{2} & , n < m \\ \frac{mn-n}{2} - n \cdot \log m + n \log n & , n \geq m \end{cases}$$

Diese Form macht auf den ersten Blick in sofern sinn, als das wir bei $n = m$ die reine Laufzeit für Selectionsort haben und bei $m = 1$ reines Mergesort benutzen. In sofern schaut die Form schoneinmal sinnvoll aus.

Als nächsten betrachten wir, wann MMergesort sich als am besten herrausstellt, wir nehmen hier wie im Tutorium vorgeschlagen, die einfache Gleichung.
Was wir wissen wollen ist, für welche $n > 0$ (vernünftige Einschränkung) gilt:

$$\begin{array}{rcl} \frac{n^2-n}{2} & < & n \cdot \log n \\ n - 1 & < & 2 \log n \\ n - \log n^2 & < & 1 \end{array}$$

Da die linke Seite monoton steigt und wir nur einen kleinen Zahlenbereich erwarten, der die Gleichung erfüllt, stellen wir einfach eine Wertetabelle auf.

n	$n - \log n^2$
1	1
2	0
3	-0.16
4	0
5	0.35
6	0.83
7	1.38

Wie man dieser Tabelle ablesen kann, ist für Listen mit eine Länge zwiswchen 1 und 6 Selectionsort besser als Mergesort. Das Minimum liegt bei 3. Daraus folgt, dass MMergesort für $m = 3$ am besten funktioniert.

(c)

Der Header mit allen wichtigen Member sieht folgender Maßen aus:

```
public class MMergesort<E extends Comparable<E>>{
  int m;

  // 2 debug member

  E[] array;
  E[] mergeArray;
```

Die Aufrufe von SORT sorgen dafür, dass array und mergeArray immer die selbe größe haben.
Unsere Implementierung von MMergesort beinhaltet 3 wichtige Funktionen.

```
private void sortHelper(){
  //First, use selection sort for parts below size of m:
  if(m > 1){
    int start = 0;
    int end = m > array.length ? array.length : m;
    while(start < array.length){
      selectionSort(start, end);
      start = start + m;
      end = end + m;
      end = end > array.length ? array.length : end;
    }
  }
  //now we begin merging the parts
  int partSize = (m > 1) ? m : 1;
  while(partSize < array.length){
    int start = 0;
    int middle = start + partSize;
    int end = middle + partSize;
    end = end > array.length ? array.length : end;
    while(start < array.length){
      if(middle>start && end>middle){
        merge(start, middle, end);
      }
      start = start + partSize;
      middle = middle + partSize;
      end = middle + partSize;
      end = end > array.length ? array.length : end;
    }
```

```
    partSize *= 2;
  }
}
```

Sorthelper führt am Anfang das Selectionsort auf den $n \mod m$ Teilstücken aus. Danach werden diese Teile Schritt für Schritt gemerged. Da wir auf einem Array arbeiten, benötigen wir den Dividestep nicht, da das Array uns schon einen linksvollständigen Baum darstellt.Je nachdem, welche 2er Potenz wir als Index und Breite nehmen, haben wir einen kompletten Unterbaum. So werden erst alle Bäume der Größe m gemerged. Danach der Größe 2m und so weiter. Wir achten bei der ausführung darauf, dass die Indizes nicht über die Arraygrenze rutschen, was bei Größen, die keine Zweiterpotenz sind passieren kann. Als nächsten wird noch betrachtet, ob m mindestens 2 ist, da sonst der Aufruf von Selectionsort ohnehin nichts tun wird.

```
private void selectionSort(int start, int end){
  int swap;
  E save;
  for(int i = start; i< end - 1; i++){
    swap = i;
    for(int j=i; j< end - 1; j++){
      compCounter++;
      if(array[j].compareTo(array[swap]) < 0) swap = j;
    }
    save = array[i];
    array[i] = array[swap];
    array[swap] = save;
  }
}
```

Das Selectionsort arbeitet In-Place. Dazu wird angenommen, das in jedem Durchlauf der äußeren Schleife links von i die Liste sortiert ist und rechts der unsortierte Teil ist. Nun wird rechts von i das kleinste Element gesucht und mit i geswapt. Soweit entspricht es der Beschreibung in 3a). Das dieser Algorithmus funktioniert wurde in ALP2 und ALP3 gezeigt.

```
private void merge(int start,int middle, int end){
  System.arraycopy(array, start, mergeArray, start, middle-
     start);
  int fst=start, snd=middle, pos = start;
  while(fst < middle && snd < end){
    if(fst == middle){
      //Copy the secend until the end
      for(int i = snd; i<end; i++){
        array[pos] = array[i];
        pos++;
      }
      return;
    }else if(snd == end){
      //Copy the first until the end
      for(int i = fst; i<middle; i++){
        array[pos] = mergeArray[i];
        pos++;
      }
      return;
    }else{
      //Wenn beides noch offen ist
      if(mergeArray[fst].compareTo(array[snd]) < 0){
        array[pos] = mergeArray[fst];
        fst++;
      }else{
```

```
        array[pos] = array[snd];
        snd++;
      }
      pos++;
    }
  }
}
```

Als erstes speichern wir den ersten Teil der Liste zwischen. Der Algorithmus arbeitet so, dass es immer die ersten beiden Elemente der beiden Listen vergleicht (fst und snd) das minimum der beiden nimmt un hinten an die neu entstehenden Liste anhängt. Sollte eine der beiden Listen leer sein, können wir den Rest der anderen jeweils komplett hinten anfügen. Das megen ended, wenn wir bei beiden Listen am Ende sind oder wenn nachdem der Rest einer der beiden Listen kopiert werden konnte.

Nachdem wir MMergesort so implementiert haben ist Mergesort leicht:

```
public class Mergesort<E extends Comparable<E>> extends
   MMergesort<E>{
  public Mergesort(){
    super(1);
  }
}
```

Da wir schon darauf geachtet haben, dass bei zu kleinem m kein Selectionsort benutzt wird, ist MMergesort mit $m = 1$ ein einfaches Mergesort.
Die Testklasse benutzt 2 Array ns und ms, die jeweils speichern, welche Werte von n und m getestet werden und probiert dann jede kombination aus.
Ein exemplarischer Testlauf sieht folgender Maßen aus (Ausgabetext gekürzt, da es sonst 2 Seiten sind):

```
---- n = 100000 -----------
Mergesort:Last sorting took '2068302 steps' and needed '49 ms'.
MMergesort(m=1): [...] '2068302 steps' [...] '21 ms'.
MMergesort(m=2):[...] '2007097 steps' [...] '20 ms'.
MMergesort(m=3):[...] '2108619 steps' [...] '19 ms'.
MMergesort(m=4):[...] '2053025 steps' [...] '17 ms'.
MMergesort(m=5):[...] '2093692 steps' [...] '17 ms'.
MMergesort(m=6):[...] '2197988 steps' [...] '19 ms'.
MMergesort(m=7):[...] '2049053 steps' [...] '16 ms'.
MMergesort(m=10):[...] '2235044 steps' [...] '17 ms'.
MMergesort(m=100):[...] '6243519 steps' [...] '39 ms'.
MMergesort(m=1000):[...] '50908051 steps' [...] '295 ms'.
---- n = 1000000 -----------
Mergesort:[...] '23923677 steps' and needed '245 ms'.
MMergesort(m=1):[...] '23923677 steps' [...] '240 ms'.
MMergesort(m=2):[...] '24050842 steps' [...] '225 ms'.
MMergesort(m=3):[...] '25139230 steps' [...] '227 ms'.
MMergesort(m=4):[...] '23835923 steps' [...] '224 ms'.
MMergesort(m=5):[...] '24747148 steps' [...] '226 ms'.
MMergesort(m=6):[...] '24978240 steps' [...] '220 ms'.
MMergesort(m=7):[...] '25692281 steps' [...] '229 ms'.
MMergesort(m=10):[...] '25958325 steps' [...] '221 ms'.
MMergesort(m=100):[...] '67818149 steps' [...] '460 ms'.
MMergesort(m=1000):[...] '513085734 steps' [...] '3014 ms'.
---- n = 10000000 -----------
Mergesort:[...] '292383010 steps' and needed '2700 ms'.
MMergesort(m=1):[...] '292383010 steps' [...] '2709 ms'.
MMergesort(m=2):[...] '290701629 steps' [...] '2553 ms'.
MMergesort(m=3):[...] '285585058 steps' [...] '2477 ms'.
MMergesort(m=4):[...] '285590369 steps' [...] '2429 ms'.
MMergesort(m=5):[...] '285706076 steps' [...] '2387 ms'.
MMergesort(m=6):[...] '293544393 steps' [...] '2430 ms'.
MMergesort(m=7):[...] '293891926 steps' [...] '2444 ms'.
MMergesort(m=10):[...] '301629925 steps' [...] '2451 ms'.
MMergesort(m=100):[...] '719346999 steps' [...] '4962 ms'.
MMergesort(m=1000):[...] '5187713975 steps' [...] '30989 ms'.
```

Wir sehen hier gerade bei den größeren Listen, das ein Wert von $m = 5$ scheinbar die beste Laufzeit hat. Das wir 1 m weiter oben liegen, wird wahrscheinlich an nicht beachteten Konstanten liegen. Dies liegt noch in unserem Interval. Man sieht auch, dass die Sortierung mit reinem Mergesort genau gleich viele Schritte benätigt, wie $m = 1$, was aber bei unserer Implementierung nicht wieter verwunderlich ist.