Max Wisniewski , Alexander Steen

Tutor: Tilmann

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:

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

Beweis

1. $2^n \in \Omega(2^{(2^n)})$:
   $\lim\limits_{n\to\infty} \frac{2^n}{2^{(2^n)}} = \lim\limits_{n\to\infty} \frac{2^n \cdot 1}{2^n \cdot 2^{(2^n)-n}} = \lim\limits_{n\to\infty} \frac{1}{2^{(2^n)-n}}$
   Da 2^n stärker wächst als n, gilt:
   $\lim\limits_{n\to\infty} \frac{1}{2^{(2^n)-n}} = 0$
   Damit gilt nach konvergenz Kriterium $2^n \in \Omega(2^{(2^n)}) \Rightarrow g_9 \in \Omega(g_{10})$
2. $n^{\log(\log(n))} \in \Omega(2^n)$:
   Es gilt: $n^{\log\log n} = 2^{\log n \cdot \log\log n} = 2^{\log(n+\log n)}$

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

   Dies ist der Fall, wenn n stärker wächst als $\log(n + \log n)$.
3. $(\lceil \log n \rceil)! \in \Omega(n^{log(\log(n))})$:
   Später
4. $4^{\log(n)} \in \Omega((\lceil \log n \rceil)!)$:
   Wie gehabt
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. $(\sqrt{2})^{\log n} \in \Theta(n^2)$:
   Ersteinmal gilt : $(\sqrt{2})^{\log n} = (2^{\frac{1}{2}})^{\log n} = 2^{\frac{1}{2}\cdot\log n} = 2^{\log\sqrt{n}} = \sqrt{2}$
   Nun wenden wir wieder das Konvergenzkreterium an:
   $\lim\limits_{n\to\infty} \frac{\sqrt{2}}{n^2} = \lim\limits_{n\to\infty} \frac{1}{n^{1.5}} = 0$
   $\Rightarrow (\sqrt{2})^{\log n} \in \Theta(n^2) \Rightarrow g_4 \in \Omega(g_5)$
7. $\log^2 n \in \Omega(\sqrt{n})$:

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

   Sagt Wolframalpha

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: $g_2 \in \Omega(g_3)$

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:
$\lim_{n\to\infty} \frac{2}{\ln n} = 0 \Rightarrow g_1 \in \Omega(g_2)$

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)

tbd

(b)

tbd

(c)

tbd

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)

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

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

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

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

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

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 $k \cdot 2^k$:

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

Setzen wir dies in unsere aufgelöste Form ein erhalten wir:

$$\begin{aligned} T(n) &= c_1 \cdot n + c_2 \cdot n \cdot \log n \\ &= 0 \cdot n + 1 \cdot n \log n \\ &= n \cdot \log n \end{aligned}$$

(b)

tbd

(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.
Mithilfe dieser konstruktion konnte eine Schranke bei etwa *MUSS NOCH GEMACHT WERDEN* gefunden werden.