Algorithmen I
Tutorium 33

Woche 5 | 25. Mai 2018

Daniel Jungkind (daniel.jungkind@student.kit.edu)

INSTITUT FÜR THEORETISCHE INFORMATIK

Inhalt

Sortieren

Insertion-/SelectionSort

Mergesort

Untere Schranke $\Omega(n \log n)$

Zu Blatt #3

Durchschnitt: etwa 68 % der Punkte

- Macht euch das Leben *leichter* und implementiert XOR-Listen und Queues **zyklisch**! ☺
- **Schusselfehler**: Spielt eure Algorithmen mal für einfache Beispiele durch

Zu Blatt #4

Durchschnitt: etwa 66 % der Punkte (mit viiieeel Gnade!)

Wenn man eine Hashtabelle (mit universeller Hashfunktion) wählt, die asymptotisch größer ist, als die Anzahl der auftretenden *insert*-Aufrufe, dauern *insert*, *remove* und *find* nur in $\mathcal{O}(1)$. **?**

Zu Blatt #4

Durchschnitt: etwa 66 % der Punkte (mit viiieeel Gnade!)

Wenn man eine Hashtabelle (mit universeller Hashfunktion) wählt, die asymptotisch größer ist, als die Anzahl der auftretenden *insert*-Aufrufe, dauern *insert*, *remove* und *find* nur in $\mathcal{O}(1)$.

Falsch.

ERWARTET $\mathcal{O}(1)$!

Zu Blatt #4

Durchschnitt: etwa 66 % der Punkte (mit viiieeel Gnade!)

Wenn man eine Hashtabelle (mit universeller Hashfunktion) wählt, die asymptotisch größer ist, als die Anzahl der auftretenden *insert*-Aufrufe, dauern *insert*, *remove* und *find* nur in $\mathcal{O}(1)$. **Falsch.**

ERWARTET $\mathcal{O}(1)$!

$\Rightarrow$ Aufgabenstellung in B (und auch C) war **kaputt**:
„Anzahl Kollisionen konstant beschränkt“ **total unrealistisch**,
Laufzeitforderung $\mathcal{O}(|R|)$ **nicht realisierbar**

$\Rightarrow$ besser: „Anzahl Kollisionen **erwartet** konstant beschränkt“
„Laufzeitforderung: **erwartet** in $\mathcal{O}(|R|)$“

$\Rightarrow$ Korrektur gnädig: Viele „null-Punkte-würdige“ Sachen kriegten trotzdem (Teil-)Punkte

Zu Blatt #4

Durchschnitt: etwa 66 % der Punkte (mit viiieeel Gnade!)

Wenn man eine Hashtabelle (mit universeller Hashfunktion) wählt, die asymptotisch größer ist, als die Anzahl der auftretenden *insert*-Aufrufe, dauern *insert*, *remove* und *find* nur in $\mathcal{O}(1)$. **Falsch.**

ERWARTET $\mathcal{O}(1)$!

$\Rightarrow$ Aufgabenstellung in B (und auch C) war **kaputt**:
„Anzahl Kollisionen konstant beschränkt“ **total unrealistisch**,
Laufzeitforderung $\mathcal{O}(|R|)$ **nicht realisierbar**
$\Rightarrow$ besser: „Anzahl Kollisionen **erwartet** konstant beschränkt“
„Laufzeitforderung: **erwartet** in $\mathcal{O}(|R|)$“

$\Rightarrow$ Korrektur gnädig: Viele „null-Punkte-würdige“ Sachen kriegten trotzdem (Teil-)Punkte

Zu Blatt #4

Durchschnitt: etwa 66 % der Punkte (mit viiieeel Gnade!)

Wenn man eine Hashtabelle (mit universeller Hashfunktion) wählt, die asymptotisch größer ist, als die Anzahl der auftretenden *insert*-Aufrufe, dauern *insert*, *remove* und *find* nur in $\mathcal{O}(1)$. **Falsch.**

ERWARTET $\mathcal{O}(1)$!

$\Rightarrow$ Aufgabenstellung in B (und auch C) war **kaputt**:
„Anzahl Kollisionen konstant beschränkt" **total unrealistisch**,
Laufzeitforderung $\mathcal{O}(|R|)$ **nicht realisierbar**
$\Rightarrow$ besser: „Anzahl Kollisionen **erwartet** konstant beschränkt"
„Laufzeitforderung: **erwartet** in $\mathcal{O}(|R|)$"

$\Rightarrow$ Korrektur gnädig: Viele „null-Punkte-würdige" Sachen kriegten trotzdem (Teil-)Punkte

SORTIEREN

Des Informatikers liebstes Hobby

Sortieralgorithmen

Ein paar Definitionen

- Ein Algorithmus heißt **in-place** $:\Leftrightarrow$ Es wird nur $O(1)$ zusätzlicher Speicher verwendet
- Ein Sortieralgorithmus heißt **stabil** $:\Leftrightarrow$ Elemente mit **exakt gleichem** Wert haben nach dem Sortieren die **gleiche Reihenfolge** zueinander wie davor

Sortieralgorithmen

Ein paar Definitionen

- Ein Algorithmus heißt **in-place** :$\Leftrightarrow$ Es wird nur $O(1)$ zusätzlicher Speicher verwendet
- Ein Sortieralgorithmus heißt **stabil** :$\Leftrightarrow$ Elemente mit **exakt gleichem** Wert haben nach dem Sortieren die **gleiche Reihenfolge** zueinander wie davor

Sortieren – Insertion-/SelectionSort

Insertion- und SelectionSort: Sortieren von Arrays

- **Idee**: Teile das Array in einen **sortierten** und einen **unsortierten** Abschnitt ein:

sortiert	unsortiert

$$\Longrightarrow$$

- Fülle schrittweise aus *unsortiert* in *sortiert*
  $\Rightarrow$ Am Ende **ganzes** Array sortiert
- Wie kann so ein Schritt aussehen?
  - **Einfügen** (*insert*) des nächsten unsortierten Elements an die korrekte Stelle im sortierten Bereich ($\Rightarrow$ *InsertionSort*)
    oder
  - **Auswählen** (*select*) des Minimums aus dem unsortierten Bereich und Anhängen ans Ende des sortierten Bereiches ($\Rightarrow$ *SelectionSort*)

Sortieren – Insertion-/SelectionSort

Insertion- und SelectionSort: Sortieren von Arrays

- **Idee**: Teile das Array in einen **sortierten** und einen **unsortierten** Abschnitt ein:

sortiert	unsortiert

$$\Longrightarrow$$

- Fülle schrittweise aus *unsortiert* in *sortiert*
$\Rightarrow$ Am Ende **ganzes** Array sortiert
- Wie kann so ein Schritt aussehen?
  - **Einfügen** (*insert*) des nächsten unsortierten Elements an die korrekte Stelle im sortierten Bereich ($\Rightarrow$ *InsertionSort*)
  oder
  - **Auswählen** (*select*) des Minimums aus dem unsortierten Bereich und Anhängen ans Ende des sortierten Bereiches ($\Rightarrow$ *SelectionSort*)

Sortieren – Insertion-/SelectionSort

Insertion- und SelectionSort: Sortieren von Arrays

- **Idee**: Teile das Array in einen **sortierten** und einen **unsortierten** Abschnitt ein:

sortiert	unsortiert

$$\Longrightarrow$$

- Fülle schrittweise aus *unsortiert* in *sortiert*
  $\Rightarrow$ Am Ende **ganzes** Array sortiert
- Wie kann so ein Schritt aussehen?
  - **Einfügen** (*insert*) des nächsten unsortierten Elements an die korrekte Stelle im sortierten Bereich ($\Rightarrow$ *InsertionSort*)
    oder
  - **Auswählen** (*select*) des Minimums aus dem unsortierten Bereich und Anhängen ans Ende des sortierten Bereiches ($\Rightarrow$ *SelectionSort*)

Sortieren – Insertion-/SelectionSort

Insertion- und SelectionSort: Sortieren von Arrays

- **Idee**: Teile das Array in einen **sortierten** und einen **unsortierten** Abschnitt ein:

sortiert	unsortiert

$$\Longrightarrow$$

- Fülle schrittweise aus *unsortiert* in *sortiert*
  $\Rightarrow$ Am Ende **ganzes** Array sortiert
- Wie kann so ein Schritt aussehen?
  - **Einfügen** (*insert*) des nächsten unsortierten Elements an die korrekte Stelle im sortierten Bereich ($\Rightarrow$ *InsertionSort*)
    oder
  - **Auswählen** (*select*) des Minimums aus dem unsortierten Bereich und Anhängen ans Ende des sortierten Bereiches ($\Rightarrow$ *SelectionSort*)

Sortieren – Insertion-/SelectionSort

Insertion- und SelectionSort: Sortieren von Arrays

- **Idee**: Teile das Array in einen **sortierten** und einen **unsortierten** Abschnitt ein:

sortiert	unsortiert

$$\Longrightarrow$$

- Fülle schrittweise aus *unsortiert* in *sortiert*
  $\Rightarrow$ Am Ende **ganzes** Array sortiert
- Wie kann so ein Schritt aussehen?
  - **Einfügen** (*insert*) des nächsten unsortierten Elements an die korrekte Stelle im sortierten Bereich ($\Rightarrow$ *InsertionSort*)
    oder
  - **Auswählen** (*select*) des Minimums aus dem unsortierten Bereich und Anhängen ans Ende des sortierten Bereiches ($\Rightarrow$ *SelectionSort*)

Sortieren – Insertion-/SelectionSort

SelectionSort

procedure SelectionSort(A : **array**[1..n] **of** Element)
  for $i := 1$ **to** n **do**
    minIndex $:= i$
    for $j := i + 1$ **to** n **do**
      if $A[j] < A[\text{minIndex}]$ **then**
        minIndex $:= j$
    // Das ausgewählte Element landet beim Index i:
    swap($A[i]$, $A[\text{minIndex}]$)

- Worst-Case?
- Best-Case?

Sortieren – Insertion-/SelectionSort

SelectionSort

procedure SelectionSort(A : **array**[1..n] **of** Element)
  for $i := 1$ **to** n **do**
    minIndex $:= i$
    for $j := i + 1$ **to** n **do**
      if $A[j] < A[\text{minIndex}]$ **then**
        minIndex $:= j$
    // Das ausgewählte Element landet beim Index i:
    swap($A[i]$, $A[\text{minIndex}]$)

- Worst-Case? $\Rightarrow \Theta(n^2)$ (immer)
- Best-Case? $\Rightarrow \Theta(n^2)$ (immer – zumindest so, wie hier, ohne Optimierungen)

Sortieren – Insertion-/SelectionSort

InsertionSort

procedure InsertionSort(A : **array**$[1..n]$ **of** Element)
    for $i := 2$ **to** n **do**
        // Füge A[i] in den sortierten Bereich ein:
        $j := i$
        while $j > 1$ **and** $A[j-1] > A[j]$ **do**
            swap($A[j-1]$, $A[j]$)
            $j{-}{-}$

- Worst-Case?
- Best-Case?

Sortieren – Insertion-/SelectionSort

InsertionSort

```
procedure InsertionSort(A : array[1..n] of Element)
    for i := 2 to n do
        // Füge A[i] in den sortierten Bereich ein:
        j := i
        while j > 1 and A[j − 1] > A[j] do
            swap(A[j − 1], A[j])
            j−−
```

- Worst-Case? $\Rightarrow \Theta(n^2)$ (umgekehrt sortiertes Array)
- Best-Case? $\Rightarrow \Theta(n)$ (bereits sortiertes Array)

Sortieren – Insertion-/SelectionSort

Das geht doch besser! Oder?

- **Idee**: Bei *InsertionSort* suchen wir die **Einfügestelle** im sortierten Teil $\Rightarrow$ **binäre Suche** anwenden (*BinaryInsertionSort*)
- Aber: **kein** (asymptotischer) **Vorteil** $\Rightarrow$ Finden der Stelle zwar in $\Theta(\log n)$, aber trotzdem noch alles rechts davon verschieben $\Rightarrow \Theta(n)$
- Immerhin: Weniger Vergleiche beim Finden $\Rightarrow$ kann sich in *manchen* Fällen trotzdem lohnen (falls Vergleiche *sehr* teuer)

Sortieren – Insertion-/SelectionSort

Das geht doch besser! Oder?

- **Idee**: Bei *InsertionSort* suchen wir die **Einfügestelle** im sortierten Teil $\Rightarrow$ **binäre Suche** anwenden (*BinaryInsertionSort*)
- Aber: **kein** (asymptotischer) **Vorteil** $\Rightarrow$ Finden der Stelle zwar in $\Theta(\log n)$, aber trotzdem noch alles rechts davon verschieben $\Rightarrow$ $\Theta(n)$
- Immerhin: Weniger Vergleiche beim Finden $\Rightarrow$ kann sich in *manchen* Fällen trotzdem lohnen (falls Vergleiche *sehr* teuer)

Sortieren – Insertion-/SelectionSort

Das geht doch besser! Oder?

- **Idee**: Bei *InsertionSort* suchen wir die **Einfügestelle** im sortierten Teil $\Rightarrow$ **binäre Suche** anwenden (*BinaryInsertionSort*)
- Aber: **kein** (asymptotischer) **Vorteil** $\Rightarrow$ Finden der Stelle zwar in $\Theta(\log n)$, aber trotzdem noch alles rechts davon verschieben $\Rightarrow$ $\Theta(n)$
- Immerhin: Weniger Vergleiche beim Finden $\Rightarrow$ kann sich in *manchen* Fällen trotzdem lohnen (falls Vergleiche *sehr* teuer)

Sortieren – Insertion-/SelectionSort

Das geht doch besser! Oder?

- **Idee**: Bei *InsertionSort* suchen wir die **Einfügestelle** im sortierten Teil $\Rightarrow$ **binäre Suche** anwenden (*BinaryInsertionSort*)
- Aber: **kein** (asymptotischer) **Vorteil** $\Rightarrow$ Finden der Stelle zwar in $\Theta(\log n)$, aber trotzdem noch alles rechts davon verschieben $\Rightarrow \Theta(n)$
- Immerhin: Weniger Vergleiche beim Finden $\Rightarrow$ kann sich in *manchen* Fällen trotzdem lohnen (falls Vergleiche *sehr* teuer)

Sortieren ...und mit Listen?

Bekannt:

- **Zwei sortierte** verkettete Listen können in $O(n)$ zu **einer sortierten** verketteten Liste zusammengefügt werden (*merge*)
- Listen der Länge 0 oder 1 sind schon **sortiert**
- Listen können **zerteilt** werden

$\Rightarrow$ Also: Zerlege Listen rekursiv bis zum Basisfall und füge die Ergebnisse jeweils zusammen $\Rightarrow$ **Mergesort**

function MergeSort(L : List of Element) : List of Element
- **if** $|L| \leq 1$ **then return** L
- **else**
  - $\begin{pmatrix} \text{first} \\ \text{second} \end{pmatrix} := \begin{pmatrix} \text{„first half" of } L \\ \text{„second half" of } L \end{pmatrix}$ // very roughly
  - **return** merge(MergeSort(first), MergeSort(second))

Sortieren ...und mit Listen?

Bekannt:

- **Zwei sortierte** verkettete Listen können in $O(n)$ zu **einer sortierten** verketteten Liste zusammengefügt werden (*merge*)
- Listen der Länge 0 oder 1 sind schon **sortiert**
- Listen können **zerteilt** werden

$\Rightarrow$ Also: Zerlege Listen rekursiv bis zum Basisfall und füge die Ergebnisse jeweils zusammen $\Rightarrow$ **Mergesort**

function MergeSort(L : List of Element) : List of Element
 if $|L| \leq 1$ **then return** L
 else
  $\begin{pmatrix} \text{first} \\ \text{second} \end{pmatrix} := \begin{pmatrix} \text{„first half" of } L \\ \text{„second half" of } L \end{pmatrix}$ *// very roughly*
  return merge(MergeSort(first),
      MergeSort(second))

Sortieren ...und mit Listen?

Bekannt:

- **Zwei sortierte** verkettete Listen können in $O(n)$ zu **einer sortierten** verketteten Liste zusammengefügt werden (*merge*)
- Listen der Länge 0 oder 1 sind schon **sortiert**
- Listen können **zerteilt** werden

$\Rightarrow$ Also: Zerlege Listen rekursiv bis zum Basisfall und füge die Ergebnisse jeweils zusammen $\Rightarrow$ **Mergesort**

function MergeSort(L : List of Element) : List of Element
  if $|L| \leq 1$ **then return** L
  else
    $\begin{pmatrix} \text{first} \\ \text{second} \end{pmatrix} := \begin{pmatrix} \text{„first half" of } L \\ \text{„second half" of } L \end{pmatrix}$ // very roughly
    return merge(MergeSort(first),
            MergeSort(second))

Sortieren – Mergesort

Bekannt:

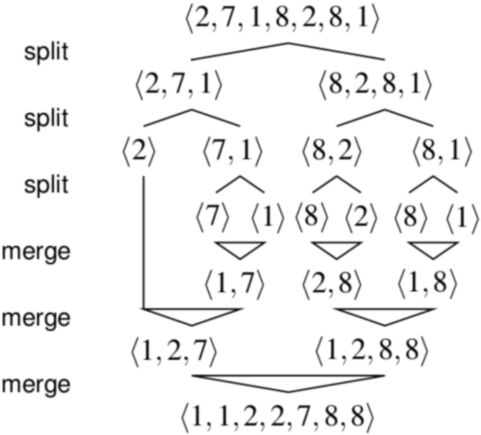
- **Zwei sortierte** verkettete Listen können in $O(n)$ zu **einer sortierten** verketteten Liste zusammengefügt werden (*merge*)
- Listen der Länge 0 oder 1 sind schon **sortiert**
- Listen können **zerteilt** werden

$\Rightarrow$ Also: Zerlege Listen rekursiv bis zum Basisfall und füge die Ergebnisse jeweils zusammen $\Rightarrow$ **Mergesort**

function MergeSort(L : List **of** Element) : List **of** Element
    if $|L| \leqslant 1$ **then return** L
    else
        $\begin{pmatrix} \text{first} \\ \text{second} \end{pmatrix} := \begin{pmatrix} \text{„first half" } \textbf{of } L \\ \text{„second half" } \textbf{of } L \end{pmatrix}$ // *very roughly*
        return merge(MergeSort(first),
                                MergeSort(second))

Sortieren – Mergesort

Schema aus der Vorlesung



Sortieren – Mergesort

Laufzeit von Mergesort

- Es ist $T(n) = \begin{cases} 1, & \text{falls } n \leqslant 1 \\ 2 \cdot T(\frac{n}{2}) + c \cdot n, & \text{falls } n > 1 \end{cases}$
- **Master-Theorem:** $T(n) \in \Theta(n \log n)$
- Input ist ein **Array**? $\Rightarrow$ Schreibe Array in eine neue verkettete Liste, wende Mergesort an, kopiere Ergebnis zurück ins Array
  ($\Rightarrow$ D. h., Mergesort für Arrays ist **nicht in-place**)

Sortieren – Mergesort

Laufzeit von Mergesort

- Es ist $T(n) = \begin{cases} 1, & \text{falls } n \leqslant 1 \\ 2 \cdot T(\frac{n}{2}) + c \cdot n, & \text{falls } n > 1 \end{cases}$
- **Master-Theorem:** $T(n) \in \Theta(n \log n)$
- Input ist ein **Array**? $\Rightarrow$ Schreibe Array in eine neue verkettete Liste, wende Mergesort an, kopiere Ergebnis zurück ins Array ($\Rightarrow$ D. h., Mergesort für Arrays ist **nicht in-place**)

Sortieren – Mergesort

Laufzeit von Mergesort

- Es ist $T(n) = \begin{cases} 1, & \text{falls } n \leqslant 1 \\ 2 \cdot T(\frac{n}{2}) + c \cdot n, & \text{falls } n > 1 \end{cases}$
- **Master-Theorem**: $T(n) \in \Theta(n \log n)$
- Input ist ein **Array**? $\Rightarrow$ Schreibe Array in eine neue verkettete Liste, wende Mergesort an, kopiere Ergebnis zurück ins Array ($\Rightarrow$ D. h., Mergesort für Arrays ist **nicht in-place**)

Sortieren – Mergesort

Laufzeit von Mergesort

- Es ist $T(n) = \begin{cases} 1, & \text{falls } n \leqslant 1 \\ 2 \cdot T(\frac{n}{2}) + c \cdot n, & \text{falls } n > 1 \end{cases}$
- **Master-Theorem**: $T(n) \in \Theta(n \log n)$
- Input ist ein **Array**? $\Rightarrow$ Schreibe Array in eine neue verkettete Liste, wende Mergesort an, kopiere Ergebnis zurück ins Array
($\Rightarrow$ D. h., Mergesort für Arrays ist **nicht in-place**)

Aufgabe: Mergesort

Aufgabe 1: Mergesort
Sortiert die Liste $\langle 65, 12, 42, 87, 5, 42, 33, 29 \rangle$ mit Mergesort und zeichnet dazu den Rekursionsbaum.

Aufgabe: Mergesort

Lösung zu Aufgabe 1

[65, 12, 42, 87, | 5, 42, 33, 29]

[65, 12, | 42, 87]   [5, 42, | 33, 29]

[65, | 12]   [42, | 87]   [5, | 42]   [33, | 29]

[65] [12] [42] [87] [5] [42] [33] [29]

[12, 65]   [42, 87]   [5, 42]   [29, 33]

[12, 42, 65, 87]   [5, 29, 33, 42]

[5, 12, 29, 33, 42, 42, 65, 87]

Sortieren

Aufgabe 2: Spaghettisort
Unterm Schrank der Vorratskammer findet ihr noch einen guten Batzen (unzubereiteter, trockener) Spaghetti. Leider sind diese im Laufe der Jahre etwas brüchig geworden und liegen daher nun in sehr vielen verschiedenen Längen vor. Um einzuschätzen, ob ihr die Spaghetti noch essen wollt oder nicht, möchtet ihr das Durcheinander in eine übersichtlichere Anordnung überführen.

Überlegt euch ein Verfahren, wie ihr die Nudeln in linearer Zeit (in Bezug auf die Anzahl der Spaghetti) der Länge nach sortieren könnt.

Sortieren

Lösung zu Aufgabe 2

Die Spaghetti in der Hand auf dem Tisch aufrichten und „absacken“ lassen. Mit der anderen Hand von oben auf die Spaghetti herabfahren und so feststellen, welche Nudel zuerst piekst. Diese ist dann die Längste und wird links zu den bereits entfernten Spaghetti dazugelegt. Wiederholen, bis keine Spaghetti mehr unsortiert sind.

Sortieren – untere Schranke

Vorlesung:

Vergleichsbasiertes Sortieren von n Elementen dauert $\Omega(n \log n)$.

Zutaten:

Sortieren – untere Schranke

Vorlesung:

Vergleichsbasiertes Sortieren von n Elementen dauert $\Omega(n \log n)$.

Zutaten:

- **Binärbaum**: Ein Baum, bei dem jeder Knoten **maximal zwei** Kindknoten besitzt.
- **Höhe** h eines Binärbaums: Die **Länge** des längsten (wiederholungsfreien) Pfades von der **Wurzel** zu einem **Blatt** (= Anzahl Kanten, über die man läuft).
- Ein Binärbaum der Höhe h hat **maximal** 2^h Blätter.

Sortieren – untere Schranke

Vorlesung:
Vergleichsbasiertes Sortieren von n Elementen dauert $\Omega(n \log n)$.

Zutaten:

- **Binärbaum**: Ein Baum, bei dem jeder Knoten **maximal zwei** Kindknoten besitzt.
- **Höhe** h eines Binärbaums: Die **Länge** des längsten (wiederholungsfreien) Pfades von der **Wurzel** zu einem **Blatt** ($=$ Anzahl Kanten, über die man läuft).
- Ein Binärbaum der Höhe h hat maximal 2^h Blätter.

Sortieren – untere Schranke

Vorlesung:

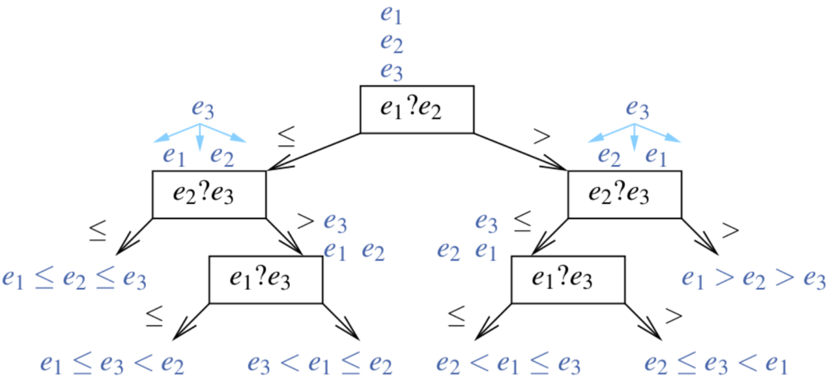
Vergleichsbasiertes Sortieren von n Elementen dauert $\Omega(n \log n)$.

Zutaten:

- **Binärbaum**: Ein Baum, bei dem jeder Knoten **maximal zwei** Kindknoten besitzt.
- **Höhe** h eines Binärbaums: Die **Länge** des längsten (wiederholungsfreien) Pfades von der **Wurzel** zu einem **Blatt** ($=$ Anzahl Kanten, über die man läuft).
- Ein Binärbaum der Höhe h hat **maximal** 2^h Blätter.

Sortieren – untere Schranke

Der Sortierbaum – Aufbau



Sortieren – untere Schranke

Der Sortierbaum – Funktionsweise

- Für jede Folgenlänge n gibt es einen **eigenen** Sortierbaum.
- Jeder **Knoten** im Baum repräsentiert den **Vergleich** zweier Elemente, dessen zwei mögliche Ergebnisse ($\leqslant$ oder $>$) zu versch. Kindern führen

$\Rightarrow$ Ganz unten: **Blätter** repräsentieren **endgültige sortierte Reihenfolge** der Elemente (die durch die vorherigen Vergleiche bekannt ist)

- Betrachte minimalen Sortierbaum T für eine Folge der Länge n.
- Der Sortierbaum muss zu *jeder möglichen Umsortierung* der Folge führen können $\Rightarrow$ er **muss** $n!$ Blätter haben.

$\Rightarrow$ Höhe $h_T \geqslant \log(n!)$

Sortieren – untere Schranke

Der Sortierbaum – Funktionsweise

- Für jede Folgenlänge n gibt es einen **eigenen** Sortierbaum.
- Jeder **Knoten** im Baum repräsentiert den **Vergleich** zweier Elemente, dessen zwei mögliche Ergebnisse ($\leqslant$ oder $>$) zu versch. Kindern führen

$\Rightarrow$ Ganz unten: **Blätter** repräsentieren **endgültige sortierte Reihenfolge** der Elemente (die durch die vorherigen Vergleiche bekannt ist)

- Betrachte minimalen Sortierbaum T für eine Folge der Länge n.
- Der Sortierbaum muss zu *jeder möglichen Umsortierung* der Folge führen können $\Rightarrow$ er **muss** $n!$ Blätter haben.

$\Rightarrow$ Höhe $h_T \geqslant \log(n!)$

Sortieren – untere Schranke

Vorlesung:

Vergleichsbasiertes Sortieren von n Elementen dauert $\Omega(n \log n)$.

Zutaten:

- **Binärbaum**: Ein Baum, bei dem jeder Knoten **maximal zwei** Kindknoten besitzt.
- **Höhe** h eines Binärbaums: Die **Länge** des längsten (wiederholungsfreien) Pfades von der **Wurzel** zu einem **Blatt** ($=$ Anzahl Kanten, über die man läuft).
- Ein Binärbaum der Höhe h hat **maximal** 2^h Blätter.
- $\log(n!) \in \Theta(n \log n)$

Sortieren – untere Schranke

$\log(n!) \in O(n \log n)$, denn:

$$\begin{aligned}\log(n!) &= \log(1 \cdot 2 \cdot \dots \cdot (n-1) \cdot n) \\ &\leqslant \log(n \cdot n \cdot \dots \cdot n \cdot n) = \log(n^n) = n \log n.\end{aligned}$$

$\log(n!) \in \Omega(n \log n)$, denn:

$$\begin{aligned}\log(n!) &= \log\left(\underbrace{1 \cdot 2 \cdots \left(\left\lfloor\frac{n}{2}\right\rfloor - 1\right)} \cdot \left\lfloor\frac{n}{2}\right\rfloor \cdot \underbrace{\left(\left\lfloor\frac{n}{2}\right\rfloor + 1\right) \cdots (n-1) \cdot n}\right) \\ &\geqslant \log\left(\overbrace{1 \cdot 1 \cdots 1 \cdot 1} \cdot \left\lfloor\frac{n}{2}\right\rfloor \cdot \overbrace{\left\lfloor\frac{n}{2}\right\rfloor \cdots \left\lfloor\frac{n}{2}\right\rfloor}\right) \\ &\geqslant \log\left(\left\lfloor\frac{n}{2}\right\rfloor^{\left\lfloor\frac{n}{2}\right\rfloor}\right) = \left\lfloor\frac{n}{2}\right\rfloor \cdot \log\left(\left\lfloor\frac{n}{2}\right\rfloor\right) \\ &\in \Theta\left(\frac{1}{2} \cdot n \cdot (\log n - \log 2)\right) = \Theta(n \log n)\end{aligned}$$

Sortieren – untere Schranke

$\log(n!) \in O(n \log n)$, denn:

$$\begin{aligned}\log(n!) &= \log(1 \cdot 2 \cdot \dots \cdot (n-1) \cdot n) \\ &\leqslant \log(n \cdot n \cdot \dots \cdot n \cdot n) = \log(n^n) = n \log n.\end{aligned}$$

$\log(n!) \in \Omega(n \log n)$, denn:

$$\begin{aligned}\log(n!) &= \log\left(\underbrace{1 \cdot 2 \cdot \dots \cdot \left(\left\lfloor \frac{n}{2} \right\rfloor - 1\right)} \cdot \left\lfloor \frac{n}{2} \right\rfloor \cdot \underbrace{\left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right) \cdot \dots \cdot (n-1) \cdot n}\right) \\ &\geqslant \log\left(\overbrace{1 \cdot 1 \cdot \dots \cdot 1 \cdot 1} \cdot \left\lfloor \frac{n}{2} \right\rfloor \cdot \overbrace{\left\lfloor \frac{n}{2} \right\rfloor \cdot \dots \cdot \left\lfloor \frac{n}{2} \right\rfloor}\right) \\ &\geqslant \log\left(\left\lfloor \frac{n}{2} \right\rfloor^{\left\lfloor \frac{n}{2} \right\rfloor}\right) = \left\lfloor \frac{n}{2} \right\rfloor \cdot \log\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \\ &\in \Theta\left(\frac{1}{2} \cdot n \cdot (\log n - \log 2)\right) = \Theta(n \log n)\end{aligned}$$

Sortieren – untere Schranke

$\log(n!) \in O(n \log n)$, denn:

$$\begin{aligned}\log(n!) &= \log(1 \cdot 2 \cdot \dots \cdot (n-1) \cdot n) \\ &\leqslant \log(n \cdot n \cdot \dots \cdot n \cdot n) = \log(n^n) = n \log n.\end{aligned}$$

$\log(n!) \in \Omega(n \log n)$, denn:

$$\begin{aligned}\log(n!) &= \log\left(\underbrace{1 \cdot 2 \cdot \dots \cdot \left(\left\lfloor \tfrac{n}{2} \right\rfloor - 1\right)} \cdot \left\lfloor \tfrac{n}{2} \right\rfloor \cdot \underbrace{\left(\left\lfloor \tfrac{n}{2} \right\rfloor + 1\right) \cdot \dots \cdot (n-1) \cdot n}\right) \\ &\geqslant \log\left(\overbrace{1 \cdot 1 \cdot \dots \cdot 1 \cdot 1} \cdot \left\lfloor \tfrac{n}{2} \right\rfloor \cdot \overbrace{\left\lfloor \tfrac{n}{2} \right\rfloor \cdot \dots \cdot \left\lfloor \tfrac{n}{2} \right\rfloor}\right) \\ &\geqslant \log\left(\left\lfloor \frac{n}{2} \right\rfloor^{\left\lfloor \frac{n}{2} \right\rfloor}\right) = \left\lfloor \frac{n}{2} \right\rfloor \cdot \log\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \\ &\in \Theta\left(\frac{1}{2} \cdot n \cdot (\log n - \log 2)\right) = \Theta(n \log n)\end{aligned}$$

Sortieren – untere Schranke

$\log(n!) \in O(n \log n)$, denn:

$$\begin{aligned}\log(n!) &= \log(1 \cdot 2 \cdot \dots \cdot (n-1) \cdot n)\\ &\leqslant \log(n \cdot n \cdot \dots \cdot n \cdot n) = \log(n^n) = n \log n.\end{aligned}$$

$\log(n!) \in \Omega(n \log n)$, denn:

$$\log(n!) = \log\left(\underbrace{1 \cdot 2 \cdot \dots \cdot \left(\left\lfloor \tfrac{n}{2} \right\rfloor - 1\right)} \cdot \left\lfloor \tfrac{n}{2} \right\rfloor \cdot \underbrace{\left(\left\lfloor \tfrac{n}{2} \right\rfloor + 1\right) \cdot \dots \cdot (n-1) \cdot n}\right)$$

$$\begin{aligned}&\geqslant \log\left(\overbrace{1 \cdot 1 \cdot \dots \cdot 1 \cdot 1} \cdot \left\lfloor \tfrac{n}{2} \right\rfloor \cdot \overbrace{\left\lfloor \tfrac{n}{2} \right\rfloor \cdot \dots \cdot \left\lfloor \tfrac{n}{2} \right\rfloor}\right)\\ &\geqslant \log\left(\left\lfloor \frac{n}{2} \right\rfloor^{\left\lfloor \frac{n}{2} \right\rfloor}\right) = \left\lfloor \frac{n}{2} \right\rfloor \cdot \log\left(\left\lfloor \frac{n}{2} \right\rfloor\right)\\ &\in \Theta\left(\frac{1}{2} \cdot n \cdot (\log n - \log 2)\right) = \Theta(n \log n)\end{aligned}$$

Sortieren – untere Schranke

$\log(n!) \in O(n \log n)$, denn:

$$\begin{aligned} \log(n!) &= \log(1 \cdot 2 \cdot \dots \cdot (n-1) \cdot n) \\ &\leqslant \log(n \cdot n \cdot \dots \cdot n \cdot n) = \log(n^n) = n \log n. \end{aligned}$$

$\log(n!) \in \Omega(n \log n)$, denn:

$$\log(n!) = \log\left(\underbrace{1 \cdot 2 \cdot \dots \cdot (\lfloor \tfrac{n}{2} \rfloor - 1)}_{\geqslant 1 \cdots 1} \cdot \lfloor \tfrac{n}{2} \rfloor \cdot \underbrace{(\lfloor \tfrac{n}{2} \rfloor + 1) \cdot \dots \cdot (n-1) \cdot n}_{\geqslant \lfloor \frac{n}{2} \rfloor \cdots \lfloor \frac{n}{2} \rfloor}\right)$$

$$\begin{aligned} &\geqslant \log\left(1 \cdot 1 \cdot \dots \cdot 1 \cdot 1 \cdot \left\lfloor \tfrac{n}{2} \right\rfloor \cdot \left\lfloor \tfrac{n}{2} \right\rfloor \cdot \dots \cdot \left\lfloor \tfrac{n}{2} \right\rfloor\right) \\ &\geqslant \log\left(\left\lfloor \frac{n}{2} \right\rfloor^{\left\lfloor \frac{n}{2} \right\rfloor}\right) = \left\lfloor \frac{n}{2} \right\rfloor \cdot \log\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \\ &\in \Theta\left(\frac{1}{2} \cdot n \cdot (\log n - \log 2)\right) = \Theta(n \log n) \end{aligned}$$

Sortieren – untere Schranke

$\log(n!) \in O(n \log n)$, denn:

$$\begin{aligned}\log(n!) &= \log(1 \cdot 2 \cdot \dots \cdot (n-1) \cdot n) \\ &\leqslant \log(n \cdot n \cdot \dots \cdot n \cdot n) = \log(n^n) = n \log n.\end{aligned}$$

$\log(n!) \in \Omega(n \log n)$, denn:

$$\begin{aligned}\log(n!) &= \log\left(\underbrace{1 \cdot 2 \cdot \dots \cdot (\lfloor \tfrac{n}{2} \rfloor - 1)}_{\geqslant 1 \cdots 1} \cdot \lfloor \tfrac{n}{2} \rfloor \cdot \underbrace{(\lfloor \tfrac{n}{2} \rfloor + 1) \cdot \dots \cdot (n-1) \cdot n}_{\geqslant \lfloor \frac{n}{2} \rfloor \cdots \lfloor \frac{n}{2} \rfloor} \right) \\ &\geqslant \log\left(\overbrace{1 \cdot 1 \cdot \dots \dots \cdot 1 \cdot 1} \cdot \lfloor \tfrac{n}{2} \rfloor \cdot \overbrace{\lfloor \tfrac{n}{2} \rfloor \cdot \dots \dots \dots \cdot \lfloor \tfrac{n}{2} \rfloor} \right) \\ &\geqslant \log\left(\left\lfloor \frac{n}{2} \right\rfloor^{\lfloor \frac{n}{2} \rfloor} \right) = \left\lfloor \frac{n}{2} \right\rfloor \cdot \log\left(\left\lfloor \frac{n}{2} \right\rfloor \right) \\ &\in \Theta\left(\frac{1}{2} \cdot n \cdot (\log n - \log 2) \right) = \Theta(n \log n)\end{aligned}$$

Sortieren – untere Schranke

$\log(n!) \in O(n \log n)$, denn:

$$\begin{aligned}\log(n!) &= \log(1 \cdot 2 \cdot \cdots \cdot (n-1) \cdot n) \\ &\leqslant \log(n \cdot n \cdot \cdots \cdot \ n \qquad \cdot n) = \log(n^n) = n \log n.\end{aligned}$$

$\log(n!) \in \Omega(n \log n)$, denn:

$$\log(n!) = \log\left(\underbrace{1 \cdot 2 \cdot \cdots \cdot (\lfloor \tfrac{n}{2} \rfloor - 1)}_{\geqslant 1 \cdots 1} \cdot \lfloor \tfrac{n}{2} \rfloor \cdot \underbrace{(\lfloor \tfrac{n}{2} \rfloor + 1) \cdot \cdots \cdot (n-1) \cdot n}_{\geqslant \lfloor \frac{n}{2} \rfloor \cdots \lfloor \frac{n}{2} \rfloor}\right)$$

$$\begin{aligned}&\geqslant \log\left(\overbrace{1 \cdot 1 \cdot \cdots \cdot 1 \cdot 1}\cdot \lfloor \tfrac{n}{2} \rfloor \cdot \overbrace{\lfloor \tfrac{n}{2} \rfloor \cdot \cdots \cdot \lfloor \tfrac{n}{2} \rfloor}\right) \\ &\geqslant \log\left(\left\lfloor \frac{n}{2} \right\rfloor^{\lfloor \frac{n}{2} \rfloor}\right) = \left\lfloor \frac{n}{2} \right\rfloor \cdot \log\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \\ &\in \Theta\left(\frac{1}{2} \cdot n \cdot (\log n - \log 2)\right) = \Theta(n \log n)\end{aligned}$$

Sortieren – untere Schranke

$\log(n!) \in O(n \log n)$, denn:

$$\log(n!) = \log(1 \cdot 2 \cdot \dots \cdot (n-1) \cdot n)$$
$$\leqslant \log(n \cdot n \cdot \dots \cdot \quad n \qquad \cdot n) = \log(n^n) = n \log n.$$

$\log(n!) \in \Omega(n \log n)$, denn:

$$\log(n!) = \log\left(\underbrace{1 \cdot 2 \cdot \dots \cdot (\lfloor \tfrac{n}{2} \rfloor - 1)}_{\geqslant 1 \cdots 1} \cdot \lfloor \tfrac{n}{2} \rfloor \cdot \underbrace{(\lfloor \tfrac{n}{2} \rfloor + 1) \cdot \dots \cdot (n-1) \cdot n}_{\geqslant \lfloor \frac{n}{2} \rfloor \cdots \lfloor \frac{n}{2} \rfloor}\right)$$
$$\geqslant \log\left(\overbrace{1 \cdot 1 \cdot \dots \cdot 1 \cdot 1} \cdot \lfloor \tfrac{n}{2} \rfloor \cdot \overbrace{\lfloor \tfrac{n}{2} \rfloor \cdot \dots \cdot \lfloor \tfrac{n}{2} \rfloor}\right)$$
$$\geqslant \log\left(\left\lfloor \frac{n}{2} \right\rfloor^{\lfloor \frac{n}{2} \rfloor}\right) = \left\lfloor \frac{n}{2} \right\rfloor \cdot \log\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$$
$$\in \Theta\left(\frac{1}{2} \cdot n \cdot (\log n - \log 2)\right) = \Theta(n \log n)$$

Sortieren – untere Schranke

Der Sortierbaum – Analyse

- Höhe $h_T \geqslant \log(n!) \in \Theta(n \log n)$
- Höhe $h_T \hat{=}$ Anzahl nötiger **Vergleiche** $\hat{=}$ (Worst-Case-)**Laufzeit**

$\Longrightarrow$ Vergleichsbasierte Sortieralgorithmen können keinen besseren Worst-Case als $\Theta(n \log n)$ haben.

$\Rightarrow$ **untere asymptotische Schranke** für vergleichsbasiertes Sortieren, „schneller geht's nicht“. $\square$

The Sound of Sorting

Karlsruher Institut für Technologie

http://panthema.net/2013/sound-of-sorting/

Sortieren

Aufgabe 3: Doktor Meta is back!

Der ebenso geniale wie hochmoderne Superbösewicht Doktor Meta ist in Aufbruchstimmung! Erst neulich hat er eine klaffende Marktlücke erkannt, mit der er seinen Reichtum mehren und schließlich die Weltherrschaft an sich reißen wird: Den Verleih von **Turingmaschinen**!

Zur Zeit besitzt Doktor Meta $k \in \mathbb{N}$ unterschiedliche Typen von TMen. Von jedem Typ $t \in \{1 \ldots k\}$ sind c_t Stück vorhanden, die alle verliehen werden können. Seine Buchungsanfragen-TM ist jedoch von seinem größten Widersacher Turing-Man (halb Mensch, halb Turingmaschine) entwendet worden, um den Superbösewicht zu stoppen.

Somit liegen also n Buchungen vor (bestehend aus *Abholzeitpunkt*, *Rückgabezeitpunkt* und *Typ* der TM) und es muss möglichst schnell (und natürlich algorithmisch) entschieden werden, ob die vorliegenden Buchungen alle erfüllt werden können.

Entwerft einen Algorithmus, der dieses Problem in höchstens $O(n \log n + k)$ löst.

(Geht davon aus, dass eine TM sofort wieder verliehen werden kann, sobald sie zurückgegeben wurde.)

Sortieren

Lösung zu Aufgabe 3

Konvertiere Buchungsliste $\rightsquigarrow$ Liste von **Ereignissen**: $\begin{pmatrix} \text{Zeitpunkt} \\ \text{Typ } t \\ dx \end{pmatrix}$,

wobei $dx = \begin{Bmatrix} -1 & \text{für Ausleihe} \\ 1 & \text{für Rückgabe} \end{Bmatrix}$.

$\Rightarrow$ Für jede Buchung werden **zwei** Ereignisse angelegt.

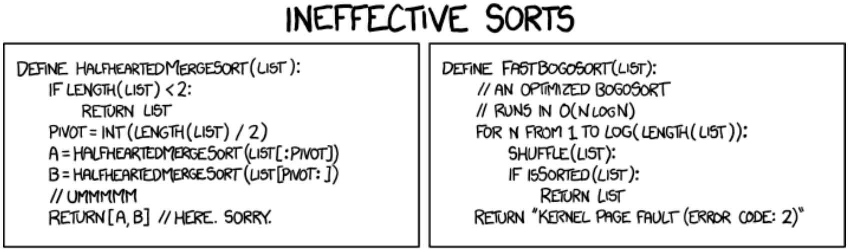
Sortiere Ereignisliste aufsteigend nach Zeitpunkt (falls gleiche Zeit $\Rightarrow$ Rückgabe **vor** Ausleihe!)

Initialisiere A : **array** $A[1...k]$ **of** $\mathbb{Z}$ mit $A[t] := c_t \quad \forall t = 1..k$.

```
for e in Ereignisse do
    A[e.t] += e.dx
    if A[e.t] < 0 then
        return false    // unerfüllbar
return true    // erfüllbar
```

Laufzeit: Sortieren von $2n$ Ereignissen: $O(n \log n)$ (mit Mergesort), Initialisieren von A in $O(k)$. Rest: $O(n) \Longrightarrow$ **insgesamt** $O(n \log n + k)$.

Danke für eure Aufmerksamkeit! ☺



http://xkcd.com/1185