Übung Algorithmen und Datenstrukturen

Sommersemester 2017

Patrick Schäfer, Humboldt-Universität zu Berlin

Agenda: Sortierverfahren

1. Schreibtischtest
2. Stabilität
3. Sortierung spezieller Arrays
4. Untere Schranke

Nächste Woche: Vorrechnen (first-come-first-served)

- Gruppe 5 13-15 Uhr https://dudle.inf.tu-dresden.de/AlgoDatGr5U2/
- Gruppe 6 15-17 Uhr https://dudle.inf.tu-dresden.de/AlgoDatGr6U2/
- Beim Vorrechnen von Programmieraufgaben bitte Source-Codes erklären.

Übung: https://hu.berlin/algodat17
Vorlesung: http://hu.berlin/vl_algodat17

Allgemeine (vergleichsbasierte) Sortierverfahren

Ein allgemeines Sortierverfahren ist ein Sortierverfahren, welches die zu sortierenden Elemente nur vergleichen kann, um die Anordnung zu bestimmen, und ansonsten keinerlei Eigenschaften der Elemente ausnutzt.

- Welche Algorithmen aus der Vorlesung sind (keine) allgemeine Sortierverfahren?

1. Selection Sort
2. Insertion Sort
3. Bubble Sort
4. Merge Sort
5. Quick Sort
6. Bucket Sort

	Comps worst case	avg. case	best case	Additional space	Moves (wc / ac)
Selection Sort	$O(n^2)$		$O(n^2)$	O(1)	O(n)
Insertion Sort	$O(n^2)$		O(n)	O(1)	$O(n^2)$
Bubble Sort	$O(n^2)$		O(n)	O(1)	$O(n^2)$
Merge Sort	O(n*log(n))		O(n*log(n))	O(n)	O(n*log(n))
QuickSort	$O(n^2)$	O(n*log(n)	O(n*log(n)	O(log(n))	$O(n^2)$ / O(n*log(n))
BucketSort (m= …)	O(m*(n+k))			O(n+k)	

Agenda

- Sortierverfahren
  1. **Schreibtischtest**
  2. Stabilität
  3. Sortierung spezieller Arrays
  4. Untere Schranke

MergeSort

Führen Sie einen Schreibtischtest für den Algorithmus **MergeSort** aus der VL für das Eingabe-Array

$$A = [x, a, b, o, k, j, c, r, g]$$

durch, wobei die Ordnung die alphabetische Ordnung auf den Buchstaben ist. Geben Sie die Zwischenschritte in Form eines Graphen wie in der VL an.

Algorithmus MergeSort(A, l, r)

Input: Array A, l linke und r rechte Grenze
Output: Sortiertes Array A

```
1: if l < r then
2:    m := (r − l) div 2;          # sortiere beide Hälften
3:    mergesort(A, l, l + m);
4:    mergesort(A, l + m + 1, r);
5:    merge(A, l, l + m, r);       # Zusammenführen der sortieren Listen
6: end if
7: return A;
```

- Zerlegt das Array in zwei gleichgroße Hälften (Zeilen 2-4).
- Fügt die sortierten Hälften im Reißverschlussverfahren zusammen (Zeile 5).

Algorithmus Merge(A, l, m, r)

Input: Array A, l, m und r Intervall-Grenzen
Output: Sortiertes Array A

```
1: B := array[1..r − l + 1];
2: i := l;                        # Anfang der 1. Liste
3: j := m + 1;                    # Anfang der 2. Liste
4: k := 1;                        # Ergebnisliste
5: while (i <= m) and (j <= r) do
6:    if A[i] <= A[j] then
7:       B[k] := A[i++];          # Aus der 1. Liste
8:    else
9:       B[k] := A[j++];          # Aus der 2. Liste
10:   end if
11:   k := k + 1;
12: end while
13: if i > m then
14:   copy A[j..r] to B[k..k + r − j];
15: else
16:   copy A[i..m] to B[k..k + m − i];
17: end if
18: copy B[1..r − l + 1] to A[l..r]; # Zurück ins Array A schreiben
19: return A;
```

QuickSort

Führen Sie einen Schreibtischtest für den Algorithmus **QuickSort** aus der VL für das Eingabe-Array

A= [2, 10, 6, 7, 13, 4, 1, 12, 5, 9]

durch, wobei Sie als Ordnung die natürliche Ordnung auf den natürlichen Zahlen annehmen. Als Pivot-Element wählen Sie das am weitesten rechts stehende Element des aktuellen Teil-Arrays.

Geben Sie den aktuellen Wert von A nach jeder swap-Operation an. Unterstreichen Sie jeweils das in diesem Aufruf von **divide**(A,l,r) betrachtete Teil-Array A[l..r].

Algorithmus QuickSort(A, l, r)

Input: Array A, l linke und r rechte Grenze
Output: Sortiertes Array A

```
1: if l < r then
2:     pos := divide(A, l, r);
3:     QuickSort(A, l, pos − 1);
4:     QuickSort(A, pos + 1, r);
5: end if
6: return A;
```

- Zerlegt das Array in zwei Hälften, basierend auf dem Pivot Element.
- Links (rechts) sind anschließend alle kleineren (größeren) Elemente.

Algorithmus divide(A, l, r)

Input: Array A, l linke und r rechte Grenze
Output: Sortiertes Array A

```
1:  val := A[r];
2:  i := l;
3:  j := r − 1;
4:  repeat
5:     while A[i] <= val and i < r do
6:        i := i + 1;                  # Element kleiner als Pivot
7:     end while
8:     while A[j] >= val and j > l do
9:        j := j − 1;                  # Element größer als Pivot
10:    end while
11:    if i < j then
12:       swap(A[i], A[j]);
13:    end if
14: until i >= j
15: swap(A[i], A[r]);
16: return i;
```

BucketSort

Bei Sortierverfahren, die *nicht* auf Vergleichen beruhen, kann eine Eingabe im Worst Case in linearer Zeit sortiert werden.

BucketSort: $O(m \cdot (n + k))$

mit k := Alphabetgröße $|\Sigma|$

m := Anzahl Stellen

n := $|A|$

- Verteilt die Elemente auf Buckets (Zeilen 3-7).
- Konkatenation der sortierten Buckets (Lines 9-14).

Algorithmus Bucketsort(A, m)

Input: Array A mit Einträgen aus Σ^m
Output: Sortiertes Array A

```
1:  B := Array leerer Queues mit |B| = |Σ|;
2:  for i := m downto 1 do      # die m-te Stelle ist die Stelle ganz rechts
3:    for j := 1 to |A| do
4:      a := A[j];
5:      k := findBucket(a, i);
6:      B[k].enqueue(a);
7:    end for
8:    j := 1;
9:    for k := 1 to |B| do
10:     while not B[k].empty() do
11:       A[j] := B[k].dequeue();
12:       j := j + 1;
13:     end while
14:   end for
15: end for
16: return A;
```

BucketSort

In dieser Teilaufgabe nutzen wir den Algorithmus **Bucketsort**, um Arrays von Zeichenketten gleicher Länge über einem festen endlichen Alphabet Σ zu sortieren. Wir nehmen also an, dass die Einträge des übergebenen Arrays alles Elemente von Σ^m für eine bestimmte Zahl $m \in \mathbb{N}_{>0}$ sind, wobei Σ^m wie üblich die Menge aller Zeichenketten über Σ der Länge m ist. Wir nehmen weiterhin an, dass das Alphabet Σ linear geordnet ist, und dass die Ordnung auf den Zeichenketten die *lexikographische* Ordnung ist. Es gilt also: $a_1 \dots a_m < b_1 \dots b_m$ g.d.w. ein i mit $1 \leq i \leq m$ existiert, so dass $a_i < b_i$ und $a_j = b_j$ für alle $j < i$. Führen Sie einen Schreibtischtest für $\Sigma = \{0, 1, 2, 3\}$, $m = 3$, und das Array

$$A \;=\; [103,\; 202,\; 101,\; 231,\; 022,\; 031,\; 030,\; 233,\; 201]$$

durch.

Notieren Sie nach jedem Durchlauf der Schleife mit Laufvariable i den Inhalt des Arrays A. Markieren Sie wie in der VL (Folie 19) mit vertikalen Strichen, welche Elemente von A sich in dieser Iteration gemeinsam in einem Bucket befanden.

Agenda

- Sortierverfahren
  1. Schreibtischtest
  2. **Stabilität**
  3. Sortierung spezieller Arrays
  4. Untere Schranke

Aufgabe (Stabilität)

Ein Sortierverfahren heißt stabil, wenn Elemente mit gleichen Schlüsseln nach der Sortierung in der gleichen Reihenfolge aufeinander folgen wie vor der Sortierung.

Entscheiden Sie, ob die folgenden Verfahren stabil sind. Falls das Verfahren nicht stabil ist, geben Sie eine (bitte möglichst kleine) Instanz als Gegenbeispiel an. Begründen Sie auf jeden Fall Ihre Entscheidung.

1. Position Sort
2. Selection Sort

Algorithmus SelectionSort(A)

Input: Array A
Output: Sortiertes Array A

```
1:  n := |A|;
2:  for i := 1 to n − 1 do
3:      min_pos := i;
4:      for j := i + 1 to n do
5:          if A[min_pos] > A[j] then
6:              min_pos := j;
7:          end if
8:      end for
9:      swap(A[min_pos], A[i]);
10: end for
11: return A;
```

- Bestimmt das kleinste Element im Rest-Array (Zeile 4-8).
- Am Ende jeder Iteration der FOR-Schleife (Zeile 1-9) steht das Minimum am Angang des Rest-Arrays.

PositionSort(S)

```
1:  n := |S|;
2:  B := neues Array der Länge n
3:  for i = 1 .. n do
4:      pos := 1;
5:      for j = 1 .. i − 1 do
6:          if S[j] ≤ S[i] then
7:              pos := pos+1;
8:          end if
9:      end for
10:     for j = i + 1 .. n do
11:         if S[j] < S[i] then
12:             pos := pos+1;
13:         end if
14:     end for
15:     B[pos] := S[i];
16: end for
17: return B;
```

- Zählt Anzahl der Elemente, die kleiner oder gleich sind.
- Am Ende jeder Iteration der FOR-Schleife (Zeile 3-16) steht das aktuelle Element an einer Position, so dass alle kleineren Elemente links und alle größeren Elemente rechts liegen.

Algorithmus InsertionSort(A)

Input: Array A
Output: Sortiertes Array A

```
1:  n := |A|;
2:  for i := 2 to n do
3:      j := i;
4:      key := A[j];
5:      while (A[j − 1] > key) and (j > 1) do
6:          A[j] := A[j − 1];
7:          j := j − 1;
8:      end while
9:      A[j] := key;
10: end for
11: return A;
```

- Fügt das i-te Element sortiert in das Array ein.
- Am Ende jeder Iteration der FOR-Schleife (Zeile 2-10) ist das Array bis zum i-ten Element sortiert.

=> Übungsaufgabe

Stabilität (PositionSort)

- Wir zeigen, dass **PositionSort** stabil ist. Wir betrachten dafür zwei Elemente s_1 und s_2 mit identischem Schlüsselwert $s_1 = s_2$.
- Falls s_1 **vor** s_2 in S steht (Zeile 10), dann hängt die Position von s_1 nicht von s_2 ab (Zeilen 11-12). Damit ist die Position von s_1 in B **kleiner** als die von s_2.
  - S=[... s_1... s_2...]
- Falls s_1 **nach** s_2 in S steht (Zeile 5), wird für die Bestimmung der Position „s_2“ mitgezählt (Zeilen 6-7). Damit ist die Position von s_1 in B **größer** als die von s_2:
  - S=[... s_2... s_1...]

PositionSort(S)

```
 1: n := |S|;
 2: B := neues Array der Länge n
 3: for i = 1 .. n do
 4:     pos := 1;
 5:     for j = 1 .. i − 1 do
 6:         if S[j] ≤ S[i] then
 7:             pos := pos+1;
 8:         end if
 9:     end for
10:     for j = i + 1 .. n do
11:         if S[j] < S[i] then
12:             pos := pos+1;
13:         end if
14:     end for
15:     B[pos] := S[i];
16: end for
17: return B;
```

Agenda

- Sortierverfahren
  1. Schreibtischtest
  2. Stabilität
  3. **Sortierung spezieller Arrays**
  4. Untere Schranke

Aufgabe (Sortierung spezieller Arrays)

Ein allgemeines Sortierverfahren ist ein Sortierverfahren, welches die zu sortierenden Elemente nur vergleichen kann und sonst keinerlei Eigenschaften der Elemente ausnutzt.

Beweisen oder widerlegen Sie: Es gibt ein *allgemeines* Sortierverfahren, welches ein Array von n beliebigen (vergleichbaren) Elementen im Worst Case in Laufzeit $O(n)$ sortiert, falls …

1. ... in einem ursprünglich sortierten Array zwei beliebige Zahl vertauscht wurden.
2. ... die ersten drei Viertel des Arrays bereits vorsortiert sind.
3. ... im Array nur Zahlen zwischen 1 und n vorkommen.

Hinweis:

- *Für den Fall, dass es ein solches allgemeines Sortierverfahren gibt, können Sie als Beweis die Idee eines konkreten Verfahrens beschreiben.*
- *Falls kein solches allgemeines Sortierverfahren existiert, können Sie die in der Vorlesung gezeigte untere Schranke für allgemeine Sortierverfahren nutzen.*
- *BucketSort ist kein allgemeines Sortierverfahren!*

Agenda

- Sortierverfahren
  1. Schreibtischtest
  2. Stabilität
  3. Sortierung spezieller Arrays
  4. **Untere Schranke**

Untere Schranke für allgemeine Sortierverfahren

Ein allgemeines Sortierverfahren kann im Worst Case nicht schneller sein als:

$$\Omega(n \log n)$$

Für ein Array mit n Elementen gibt es n! mögliche Permutationen. In einem Entscheidungsbaum mit $n!$ Blättern, der vollständig balanciert ist, hat ein Blatt eine minimale Tiefe von $h \geq \log\ (n!)$:

$$h \geq \log(n!) \geq \frac{n}{2} \log \frac{n}{2} \in\ \Omega(n \log \mathrm{n})$$

Hinweis:

$$\begin{aligned} log\, n! \quad &= log(1)+log(2)+\ldots+log(n-1)+log(n) \\ &\geq log\left(\frac{n}{2}\right)+log\left(\frac{n}{2}+1\right)+\ldots+log(n) \\ &\geq log\left(\frac{n}{2}\right)+log\left(\frac{n}{2}\right)+\ldots+log(\frac{n}{2}) \\ &= \frac{n}{2}\ log\frac{n}{2} \end{aligned}$$

Nächste Woche

Nächste Woche: Vorrechnen (first-come-first-served)

- Gruppe 5 13-15 Uhr https://dudle.inf.tu-dresden.de/AlgoDatGr5U2/
- Gruppe 6 15-17 Uhr https://dudle.inf.tu-dresden.de/AlgoDatGr6U2/
- Beim Vorrechnen von Programmieraufgaben bitte Source-Codes erklären.

Übung: https://hu.berlin/algodat17

Vorlesung: http://hu.berlin/vl_algodat17