DA-1

Lorenz Pfäffli 18-925-230
Lukas Ingold 20-123-998

Datenstrukturen und Algorithmen
Übung 01 – Grundlagen

Abgabefrist: 04.03.2020, 16:00 h

Verspätete Abgaben werden nicht bewertet.

Ihre Lösung der theoretischen Aufgaben im PDF-Format, der gesamte Sourcecode + Ausgaben ihrer Programme (.txt) müssen elektronisch über Ilias abgegeben werden.

Die Übung sollte in Zweiergruppen bearbeitet werden[a]. Sie können das Forum im Ilias nutzen, um Gruppenpartner zu finden[b].

Jede Übungsserie gibt 10 Punkte. Im Durchschnitt müssen Sie 7 von 10 Punkten erreichen, um die Testatbedingung zu erfüllen und an der Prüfung teilzunehmen.

[a] bei einer ungeraden Anzahl an Studierenden ist natürlich eine Einzel- oder Dreier-Gruppe nötig
[b] Bitte editieren oder löschen Sie ihren Post, nachdem Sie eine Parter*in für Ihre Gruppe gefunden haben

Grundlagen: Summen, Produkte und Induktion

1. Leiten Sie eine einfache Formel für $\sum_{k=1}^{n}(5k+1)$ her. **(1 Punkt)**
2. Werten Sie das Produkt $\Pi_{k=1}^{n} 3 \cdot 2^k$ aus. **(1 Punkt)**
3. Zeigen Sie, dass die Reihe $\sum_{k=1}^{n}\frac{1}{k^2}$ durch eine von n unabhängige Konstante nach oben beschränkt ist. Beweisen Sie dies aber nicht direkt, sondern beweisen Sie zu erst folgendes Hilfsresultat per Induktion und benutzen Sie es um die Aussage zu beweisen:

$$\sum_{k=1}^{n}\frac{1}{k^2} \leq 1+1-\frac{1}{n}$$

Strukturieren Sie den induktiven Beweis so, wie in der Vorbesprechung vorgestellt. Es könnte ausserdem nützlich sein, dass $\frac{1}{n} = \frac{1}{n+1} + \frac{1}{n(n+1)}$; dies gilt wegen $\frac{1}{n} - \frac{1}{n+1} = \frac{n+1-n}{n(n+1)} = \frac{1}{n(n+1)}$.
(1 Punkt)

Theoretische Aufgaben

1. Illustrieren Sie den Ablauf von Sortieren durch Einfügen (INSERTION SORT) anhand einer Skizze. Verwenden Sie die Eingabe $\langle 15,5,16,11,2,1,8,3,6,2\rangle$. **(1 Punkt)**
2. Illustrieren Sie den Ablauf von Sortieren durch Mischen (MERGE SORT) ähnlich wie in Figure 2.4 im Buch. Verwenden Sie dieselbe Eingabe wie oben. Wird ein Feld $A[p...r]$ ungerader Länge in zwei Hälften geteilt soll die untere Hälfte ein Element grösser sein als die obere. **(1 Punkt)**

3. Wir betrachten das folgende *Suchproblem*:
Eingabe: Ein Feld von Zahlen $A = \langle a_1, a_2, \dots a_n\rangle$ und ein Wert v.
Ausgabe: Ein Index i, so dass $v = A[i]$ oder den speziellen Wert NIL, falls v nicht in A vorkommt.
   (a) Schreiben Sie Pseudocode, welcher das Problem löst, indem das Feld Element um Element durchlaufen wird, bis v gefunden oder das Ende des Feldes erreicht wird.
   (b) Zeigen Sie, dass Ihr Algorithmus korrekt ist, indem Sie eine Schleifeninvariante beweisen. Das heisst: formulieren Sie eine Schleifeninvariante, „beweisen“ sie die Invariante, indem Sie je in einigen Sätzen die Initialisierungseigenschaft, die Fortsetzungseigenschaft und die Terminierungseigenschaft begründen. Benutzen Sie dann die Schleifeninvariante um die Korrektheit des Algorithmus zu beweisen. (Vgl. auch Kapitel 2.1 im Buch)
   (c) Was ist die Worst-Case-Laufzeit dieses Algorithmus?
   (1 Punkt)
4. Das Suchproblem oben kann effizienter gelöst werden, wenn das Eingabefeld sortiert ist. In diesem Fall kann das Element in der Mitte des Feldes mit v verglichen werden, und die eine Hälfte des Feldes kann von der Suche ausgeschlossen werden. BINÄRE SUCHE ist ein Algorithmus, der diese Idee wiederholt anwendet, indem die verbleibende Hälfte des Feldes jeweils wieder halbiert wird.
   (a) Schreiben Sie Pseudocode für binäre Suche, der iterativ (nicht rekursiv) abläuft.
   (b) Zeigen Sie, dass Ihr Algorithmus korrekt ist, indem Sie eine Schleifeninvariante beweisen.
   (c) Erklären Sie, warum die Worst-Case-Laufzeit des Verfahrens $\Theta(\log n)$ ist.
   (1 Punkt)

Praktische Aufgaben

1. In der Vorlesung wurden für INSERTION SORT die Zeitkomplexität $\mathcal{O}(n^2)$ und für MERGE SORT $\mathcal{O}(n \log n)$ angegeben. Bestätigen Sie diese Grössenordnungen experimentell. Sie finden Java-Code für beide Algorithmen sowie eine Timer-Klasse im .zip dieser Aufgabenstellung.

Schreiben Sie ein Java Programm zur Durchführung des Experiments. Erzeugen Sie zufällige Eingabefelder verschiedener Grösse und messen Sie jeweils die Zeit, die zum Sortieren nötig ist. Zur Zeitmessung können Sie die Klasse Timer verwenden, die wir zur Verfügung stellen. Beachten Sie, dass die Messung von kurzen Zeitspannen relativ ungenau sein kann.

Als Abgabe erwarten wir den Ausdruck Ihres Testprogramms. Weiter sollen Sie eine Grafik erstellen, welche die Zeitmessungen für beide Algorithmen und verschieden grosse Felder zusammenfasst. Erwähnen Sie auch die Spezifikationen des Computers, auf denen das Experiment durchgeführt wurde. Erläutern Sie, ob Ihre Messungen der Theorie entsprechen.

Schätzen Sie zudem auf Grund Ihrer Messungen die Laufzeit von INSERTION SORT bei der Eingabe von 10'000'000 Zahlen ab.

Hinweise: Konsultieren Sie die Java-Dokumentation, um herauszufinden wie Zufallszahlen erzeugt werden können. Sie können die Grafik (akkurat) von Hand zeichnen oder ein Softwareprogramm wie z.B. gnuplot, R, oder Excel verwenden.

Vergessen Sie nicht ihren Sourcecode innerhalb der Deadline über die Ilias-Aufgabenseite einzureichen.
(3 Punkte)

Grundlagen: Summen, Produkte, Induktion

1) $\sum_{k=1}^{n}(5k+1) = 5\cdot\sum_{k=1}^{n}k + \sum_{k=1}^{n}1$
$= 5\cdot\frac{1}{2}n(n+1) + n$
$= \frac{5}{2}n(n+1) + n$
$= \frac{5}{2}n^2 + \frac{5}{2}n + n$
$= \frac{5}{2}n^2 + \frac{7}{2}n$
$= \frac{1}{2}n(5n+7)$

$\sum_{k=1}^{n}k = \frac{1}{2}n(n+1)$
$\prod_{k=1}^{n}k = n!$

2) $\prod_{k=1}^{n}3\cdot 2^k = 3^n\cdot\prod_{k=1}^{n}2^k$
$= 3^n\cdot 2^1\cdot 2^2\cdot 2^3\cdot 2^4\cdot\dots$
$= 3^n\cdot 2^{1+2+3+4+\dots}$
$= 3^n\cdot 2^{\frac{1}{2}n(n+1)}$

(3) Beweise per Induktion $S_n \Leftrightarrow \sum_{k=1}^{n}\frac{1}{k^2} \leq 1+1-\frac{1}{n}$

Induktionsverankerung (S_1)
$1 \leq 1+1-1$
$1 \leq 1$
$\Rightarrow S_1$ ist wahr

Induktionsannahme:
Die zu beweisende Aussage S_n stimmt für gewisse $n \in \mathbb{N}$

Induktionsschritt: Wir zeigen, dass $S_{n'}$ für $n' = n+1$ wahr ist.

$\sum_{k=1}^{n+1}\frac{1}{k^2} \leq 1+1-\frac{1}{n+1}$

$\underbrace{\sum_{k=1}^{n}\frac{1}{k^2}}_{\leq(1+1-\frac{1}{n})} + \frac{1}{(n+1)^2} \leq 1+1-\frac{1}{n+1}$

$\sum_{k=1}^{n}\frac{1}{k^2} + \frac{1}{(n+1)^2} \leq 1+1-\frac{1}{n}+\frac{1}{(n+1)^2} \leq 1+1-\frac{1}{n+1}$ | -2
$-\frac{1}{n}+\frac{1}{(n+1)^2} \leq -\frac{1}{n+1}$ | $\cdot(-1)$
$\frac{1}{n}-\frac{1}{(n+1)^2} \geq \frac{1}{n+1}$ | $\cdot(n+1)$
$\frac{(n+1)}{n}-\frac{1}{(n+1)} \geq 1$ | $\cdot n$
$n+1-\frac{n}{n+1} \geq 1$ | -1
$n-\frac{n}{n+1} \geq 0$ | $+\frac{n}{n+1}$
$n \geq \frac{n}{n+1}$
$n(n+1) \geq n$

$\Rightarrow$ Diese Ungleichung ist korrekt. Da sie durch äquivalente Umformungen herbeigeführt worden ist stimmt $S_{n'}$

$\Rightarrow$ Somit wurde die Aussage S_n per Induktionsprinzip für alle $n \in \mathbb{N}$ bewiesen.

$\Leftrightarrow \forall n \in \mathbb{N}: \sum_{k=1}^{n}\frac{1}{k^2} \leq 1+1-\frac{1}{n}$

Theoretische Aufgaben (3)

a)
```
suche(A, v) {
1  for i = 1 to A.length      // Wir benutzen Arrayindizes die mit 1 beginnen
2     if (A[i] == v) {
3         return i;
4     }
5  }
6  return NIL;
}
```

$\forall x \quad x < i : A[x] \neq v$

$A[i \dots n]$

b)
Schleifeninvariante:
Für alle $x \in [1, A.length]$ mit $x < i$ gilt: Das Element $A[x]$ ist nicht v.
$\forall x$ mit $1 \leq x < i : A[x] \neq v$

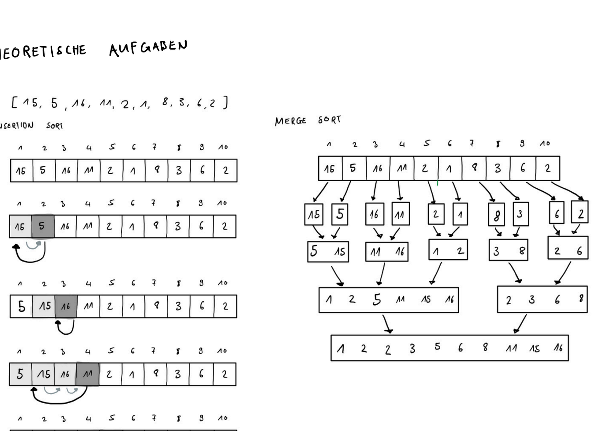
Initialisierungseigenschaft:
Der Loopindex wird mit $i = 1$ initialisiert. Das Subarray welches wir betrachten entspricht der leeren Menge. $\{x \mid x \in [1, A.length]$ mit $x < i\} = \emptyset$
Die Schleifeninvariante trifft zu, da wir keine Elemente betrachten.

Fortsetzungseigenschaft:
Wir gehen davon aus, dass die Invariante vor der Iteration zutrifft. Wenn das Element $A[i]$ ungleich v ist, dann trifft die Invariante auch vor der nächsten Iteration zu. Dies zeigt die folgende Implikation:
$(\forall x$ mit $1 \leq x < i : A[x] \neq v)$ und $(A[i] \neq v) \Rightarrow (\forall x$ mit $1 \leq x < (i+1) : A[x] \neq v)$

Wenn $A[i]$ gleich v ist, dann wird die Momentane Iteration beendet.

Terminierungseigenschaft:
Die for-Schleife wird terminiert entweder wenn $A[i] = v$ ist, oder wenn $i > A.length$ ist.
Im ersten Fall trifft die Invariante zu.
Im zweiten Fall trifft sie auch zu, da i beim letzten Test der Loopcondition $A.length + 1$ entspricht.
$(\forall x$ mit $1 \leq x \leq A.length : A[x] \neq v)$

c) Im schlimmsten Fall werden alle Elemente von A betrachtet. Somit ist die Worst-Case-Laufzeit ist linear bzw $O(n)$.



(4.a)

```
Binäre Suche(A, v) {
1  l = 1                  // linke Schranke
2  r = A.length           // rechte Schranke
3  while (l ≤ r) {
4      m = ⌊(r+l)/2⌋      // mitte
5      if (A[m] == v) { return m }
6      else if (A[m] > v) { r = m - 1 }
7      else { l = m + 1 }
8  }
9  return NIL;
}
```

(b)
Schleifeninvariante:
Falls $v \in A[1 \dots A.length]$: $A[l] \leq v \leq A[r]$

Initialisierungseigenschaft:
Da l mit 1 und r mit A.length initialisiert werden trifft die Invariante zu.

Fortsetzungseigenschaft:
Wir gehen davon aus, dass die Invariante vor der Schleifeniteration erfüllt ist.
Fallunterscheidung:
Wenn $(A[m] = v)$, dann terminiert der Algorithmus. (Deshalb ist dieser Fall für die Fortsetzungseigenschaft nicht relevant)

Wenn $(A[m] > v)$ dann wird die rechte "Schranke" r mit $(m-1)$ ersetzt. Da A sortiert ist, trifft die Invariante zu.

Wenn $(A[m] < v)$ dann wird die linke "Schranke" l mit $(m+1)$ ersetzt. Da A sortiert ist, trifft die Invariante zu.

Terminierungseigenschaft:
Wir betrachten die Fälle, in denen der Algorithmus terminiert. (Zeile 5, 9)
Wenn $A[m]$ gleich v ist, dann bleibt die Invariante erfüllt, da m immer entweder mit $(m+1)$ oder $(m-1)$ ersetzt worden ist.
Falls Zeile 9 erreicht wird, ist die Invariante auch erfüllt, da sie voraussetzt, dass v im Array A enthalten sein muss.

4.c) Aus dem Pseudocode lässt sich folgende Rekurrenzgleichung ableiten.

$$T(n) = \begin{cases} T(n/2) + c_2 & \text{falls } n > 1 \\ c_1 & \text{falls } n = 1 \end{cases}$$

(c_1, c_2 sind Konstanten. n ist die Länge des Arrays A.)

Die Länge des zu betrachtenden Arrays nach jeder Schleifeniteration halbiert.
Im schlimmsten Fall hat das letzte Subarray die Länge 1.

$1 = \frac{n}{2^x}$
$2^x = n$
$\log_2(2^x) = \log_2(n)$
$x\cdot\log_2(2) = \log_2(n)$
$x = \log_2(n)$

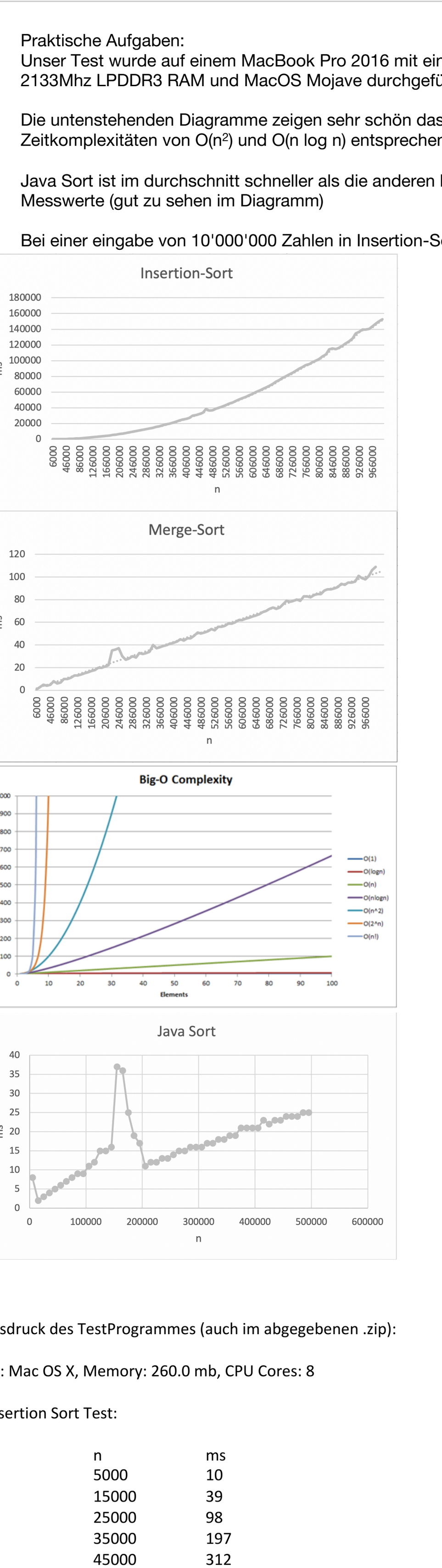
x entspricht der Anzahl Schleifeniterationen
Somit liegt die Laufzeit der binären Suche in $\Theta(\log(n))$.

Praktische Aufgaben:
Unser Test wurde auf einem MacBook Pro 2016 mit einem Intel i7 2.9Ghz 8 Core Prozessor,16GB 2133Mhz LPDDR3 RAM und MacOS Mojave durchgeführt.

Die untenstehenden Diagramme zeigen sehr schön das die Messwerte den erwarteten Zeitkomplexitäten von $O(n^2)$ und $O(n \log n)$ entsprechen.

Java Sort ist im durchschnitt schneller als die anderen beiden allerdings gab es bei uns verfälschte Messwerte (gut zu sehen im Diagramm)

Bei einer eingabe von 10'000'000 Zahlen in Insertion-Sort rechnen wir mit einer Dauer von ca.: 2.77h



Ausdruck des TestProgrammes (auch im abgegebenen .zip):

OS: Mac OS X, Memory: 260.0 mb, CPU Cores: 8

Insertion Sort Test:

id	n	ms
1	5000	10
2	15000	39
3	25000	98
4	35000	197
5	45000	312
6	55000	450
7	65000	663
8	75000	886
9	85000	1109
10	95000	1375
11	105000	1715
12	115000	2071
13	125000	2405
14	135000	2888
15	145000	3218
16	155000	3558
17	165000	3941
18	175000	4455
19	185000	5019
20	195000	5848
21	205000	6420
22	215000	6695
23	225000	7385
24	235000	8089
25	245000	8730
26	255000	9546
27	265000	10812
28	275000	11030
29	285000	11871
30	295000	12849
31	305000	13412
32	315000	14942
33	325000	15549
34	335000	16284
35	345000	17314
36	355000	18557
37	365000	19643
38	375000	22170
39	385000	22392
40	395000	22741
41	405000	24105
42	415000	25070
43	425000	28190
44	435000	30000
45	445000	31158
46	455000	33479
47	465000	34447
48	475000	35842
49	485000	38009
50	495000	39083

Merge Sort Test:

id	n	ms
1	5000	2
2	15000	3
3	25000	6
4	35000	7
5	45000	5
6	55000	6
7	65000	7
8	75000	10
9	85000	12
10	95000	14
11	105000	15
12	115000	11
13	125000	14
14	135000	14
15	145000	17
16	155000	22
17	165000	24
18	175000	26
19	185000	28
20	195000	26
21	205000	22
22	215000	22
23	225000	24
24	235000	24
25	245000	26
26	255000	28
27	265000	27
28	275000	29
29	285000	30
30	295000	33
31	305000	33
32	315000	32
33	325000	33
34	335000	34
35	345000	36
36	355000	37
37	365000	37
38	375000	38
39	385000	42
40	395000	40
41	405000	41
42	415000	46
43	425000	44
44	435000	45
45	445000	47
46	455000	46
47	465000	49
48	475000	50
49	485000	50
50	495000	51

Java Sort Test:

id	n	ms
1	5000	8
2	15000	2
3	25000	3
4	35000	4
5	45000	5
6	55000	6
7	65000	7
8	75000	8
9	85000	9
10	95000	9
11	105000	11
12	115000	12
13	125000	15
14	135000	15
15	145000	16
16	155000	37
17	165000	36
18	175000	25
19	185000	19
20	195000	17
21	205000	11
22	215000	12
23	225000	12
24	235000	13
25	245000	13
26	255000	14
27	265000	15
28	275000	15
29	285000	16
30	295000	16
31	305000	16
32	315000	17
33	325000	17
34	335000	18
35	345000	18
36	355000	19
37	365000	19
38	375000	21
39	385000	21
40	395000	21
41	405000	21
42	415000	23
43	425000	22
44	435000	23
45	445000	23
46	455000	24
47	465000	24
48	475000	24
49	485000	25
50	495000	25

average time and n for Insertion-Sort:

12721 ms 250000

average time and n for Merge-Sort:

27 ms 250000

average time and n for Java-Sort:

16 ms 250000