Lösungshinweise

Test 5 - Such- und Sortieralgorithmen

Modul prog, WS23/24
Trier University of Applied Sciences
Informatik Fernstudium (M.C.Sc.)

23.01.2024
Thorsten Suckow-Homberg, `https://thorsten.suckow-homberg.de`

Das Abrufdatum aller in diesem Dokument aufgeführten Webseiten war der 22.01.2024.

Die Lösungshinweise beinhalten nicht die Testaufgaben, bei denen eine bestimmte Programmausgabe nachvollzogen werden musste.

Um Konzepte des Lehrmaterials deutlicher von weiterführenden, für den Kurs weniger relevante Themen abzugrenzen, sind diese nun unter **Exkurs** in den jeweiligen Abschnitten zusammengefasst.

Inhaltsverzeichnis

1

Anzahl der Vergleiche

Lösung

Eine Laufzeitabschätzung führt zu $O(n^{\frac{4}{3}})$ $(= O(n^{1.3}))$.

Anmerkungen und Ergänzungen

Bei dem Algorithmus handelt es sich um einen Sortieralgorithmus, der bereits 1959 von *Donald Shell*[1] vorgestellt wurde, und mit **Shellsort** auch nach ihm benannt wurde.

Die in dem Algorithmus verwendete Sortiermethode ist auch bekannt als *Sortieren mit abnehmenden Inkrementen* [OW17b, 88], das Verfahren ist eine Variation von **Insertion Sort**:

> [Shellsort] uses insertion sort on periodic subarrays of the input to produce a faster sorting algorithm. ([CL22, 48])

In der vorliegenden Implementierung werden $t = log_2(n)$ Inkremente[2] h_t[3] der Form $\frac{n}{2^i}$ verwendet, um $lg(n)$ h-sortierte Folgen zu erzeugen. Im letzten Schritt sortiert der Algorithmus dann in h_1 die Schlüssel mit Abstand=1.

Die Effizienz des Sortierverfahrens ist stark abhängig von h: So zeigt *Knuth*, dass $O(n^{\frac{3}{2}})$ gilt, wenn für h gilt: $h_s = 2^{s+1} - 1$ mit $0 \leq s < t = lg(n)$ (vgl. [Knu97, 91])[4]. In unserem Fall können wir von $O(n^2)$ ausgehen.

---

[1] [She59]

[2] mit $n =$ Länge des zu sortierenden Feldes. Im folgenden lg für log_2.

[3] vgl. [Knu97, 84]

[4] Knuth bezieht sich hier auf die Arbeit von *Papernov und Stasevich* ([PS65]). Weitere Verweise auf Laufzeiten in Abhängigkeit von n und h fasst übersichtlich der Wikipedia-Artikel zusammen: `https://en.wikipedia.org/wiki/Shellsort` auf.

Aufgabenstellung

In der Aufgabe wurde nach der Anzahl der zwischen zwei Feldelementen gemachten Vergleiche gefragt (`feld[j] > feld[j + distanz]`), die für große randomisierte Felder durchgeführt werden.

Um die Aufrufe zu zählen, kann wie in Listing 1.1 eine Zählvariable $c4$ eingesetzt wird, die jeden Aufruf über ein *increment* protokolliert.

```
1
2  while (distanz > 0) {
3      c1++;
4      for (i = distanz; i < feld.length; i++) {
5          j = i - distanz;
6          c2++;
7          while (j >= 0) {
8              c3++;
9              if (feld[j] > feld[j + distanz]) {
10                 c4++;
11                 swap(feld, j, j+distanz);
12                 j = j - distanz;
13             } else {
14                 j =-1;
15             }
16         }
17     }
18     distanz = distanz / 2;
19 }
```

Listing 1.1. Zur Protokollierung der Aufrufe können in dem Code Zählvariablen eingesetzt werden ($c1, .., c4$).

Zur Lösung der Aufgabe können randomisierte Felder erzeugt werden, wie es der Test-Code in Listing 1.2 demonstriert. Dann wird überprüft, in welchen Bereichen sich $c4$ bewegt. Da jeder auf Vergleichsoperationen basierende Sortieralgorithmus zu der Laufzeitklasse $\Omega(n\ log\ n)$ gehört[5], kann $c4 \geq n\ log\ n$ angenommen werden:

> Jedes allgemeine Sortierverfahren benötigt zum Sortieren von N verschiedenen Schlüsseln sowohl im schlechtesten Fall als auch im Mittel wenigstens $\Omega(N\ log\ N)$ Schlüsselvergleiche. ([OW17b, 154, Satz 2.4])

Allerdings wird $n^{1.3}$ ab $n = 982$ schneller wachsen als $n\ log\ n$ (siehe Abbildung 1.1), weshalb das in der Auswertung berücksichtigt werden muss - man könnte sonst fälschlicherweise meinen, das in dem geg. Fall die Laufzeit im Durchschnitt $n\ log\ n$ beträgt.

[5] [OW17b, 154], Skript (Teil2) S. 180

```
1
2  int epochs = 2000;
3  while(epochs-- >= 0) {
4      int[] tests = new int[]{10, 100, 1000, 100_000, 1_000_000};
5      java.util.Random r = new java.util.Random();
6      for (int i = 0; i < tests.length; i++) {
7          int l = tests[i];
8          int[] arr = new int[l];
9
10         for (int j = l; j > 0; j--) {
11             arr[l - j] = r.nextInt(l + 1);
12         }
13
14         ShellSort.sort(Arrays.copyOfRange(arr, 0, arr.length));
15     }
16 }
```

Listing 1.2. Code zum Erzeugen randomisierter Felder zum Testen von Shellsort.

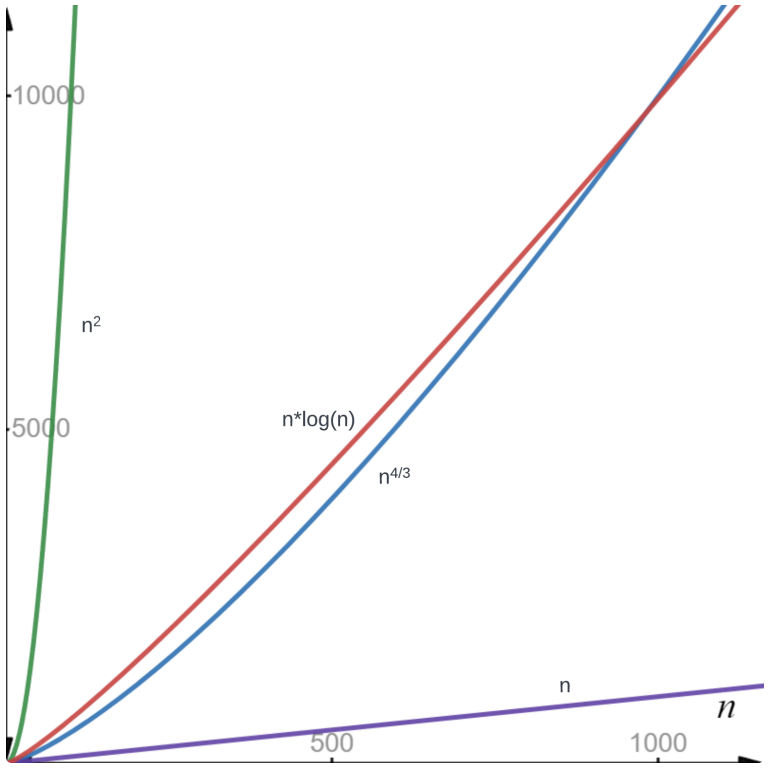

Abb. 1.1. Ab $n = 982$ wächst $n^{1.3}$ schneller als $n\ log\ n$.

Wegen $\Omega(n\ log\ n)$ interessieren wir uns für die Intervalle[6] :

- $n\ log\ n \leq c4 \leq n^{1.3}$
- $n\ log\ n \leq n^{1.3} \leq c4 \leq n^2$

Die Ausführung des Tests zeigt, dass die Aufrufanzahl von $c4$ bei großen n unterhalb $n^{1.3}$ oder zwischen $n^{1.3}$ und n^2 liegt:

[6] siehe dazu auch Tabelle 1.1

It appears that the time required to sort n elements is proportional to $n^{1.226}$.[7] ([She59, 31])

Laufzeitanalyse

Für die nachfolgenden Betrachtungen sei eine Eingabegröße $n = 2^p$ gegeben. Die Inkremente h entsprechen der über die Implementierung vorgegebenen Folge h mit $h_t = \frac{n}{2}, h_{t-1} = \frac{n}{4}, \ldots, h_1 = 1$.

Für die Analyse ist die **Shellsort**-Implementierung in Listing 1.3) gegeben.

```
1
2  public static int[] sort(int[] arr) {
3
4      int n = arr.length;
5      int delta = n/2;
6      int min;
7      int j;
8
9      while (delta > 0) { // c1
10
11         for (int i = delta; i < arr.length; i++) {
12
13             // c2
14
15             min = arr[i];
16             j = i;
17
18             while (j - delta >= 0 && min < arr[j - delta]) {
19
20                 // c3;
21
22                 arr[j] = arr[j - delta];
23                 j -= delta;
24             }
25             arr[j] = min;
26         }
27
28         delta = delta / 2;
29
30     }
31     return arr;
32 }
```

Listing 1.3. Implementierung des Shellsort-Algorithmus.

Im Folgenden betrachten wir die Anzahl für die im Code durch Kommentare markierten Stellen $c1$ in Zeile 11, $c2$ in Zeile 13 und $c3$ in Zeile 20.

Für eine erste *worst-case*-Analyse ist ein Feld der Länge 16 gegeben, in absteigender Reihenfolge sortiert (s. Abbildung 1.2)

16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

Abb. 1.2. Die für die Laufzeitabschätzung verwendete Eingabefolge 16..1

[7] *Shell* führt in seinem Paper keinen Beweis auf. Er stützt seine Behauptung auf Messungen, die er selber in Tests durchgeführt hat: "[...] an analytical determination of the expected speed has eluded the writer. However, experimental use has established its speed characteristics." (ebenda)

Ganz offensichtlich gilt für **c3**, dass es $lg(n)$-mal aufgerufen wird[8].

c2 befindet sich im Block der durch die in Zeile 11 definierten Zählschleife.
Der Startwert für `i` ist in jedem Durchgang des Blocks $c1$ der aktuelle Wert von `delta`[9], und läuft jeweils bis $n-1$.
In einem kompletten Durchlauf der Schleife enstpricht die Anzahl der Aufrufe von $c2$ also

$$(n-1) - delta + 1 = n - delta \tag{1.1}$$

Hinweis:
Für das Beispiel betrachten wir der Einfachheit halber Feldlängen der Form 2^p.
Für den allgemeinen Fall gilt für die Anzahl der Aufrufe von $c2$:

$$\sum_{i=1}^{\lfloor lg(n) \rfloor} n - \lfloor \frac{n}{2^i} \rfloor = n * lg(n) - \sum_{i=1}^{\lfloor lg(n) \rfloor} \lfloor \frac{n}{2^i} \rfloor$$

Für die Gesamtzahl der Aufrufe von $c2$ ergibt sich somit unter Berücksichtigung von $c1$

$$\sum_{i=1}^{lg(n)} n - \frac{n}{2^{i-1}} \tag{1.2}$$

was nach Auflösen

$$n * lg(n) - n + 1 \tag{1.3}$$

entspricht, und für unser Beispiel

$$16 * lg(16) - 16 + 1 = 49 \tag{1.4}$$

ist.
In dem durch die in Zeile 18 definierte `while`-Schleife findet die eigentliche Arbeit des Algorithmus statt: Es wird überprüft, ob *delta*-entfernte Elemente in aufsteigender Reihenfolge sortiert angeordnet sind.
Ist das nicht der Fall, werden die Elemente an den Stellen j und $j - delta$ ausgetauscht, bis die h-Folge sortiert ist.
Für den ersten Durchgang des Algorithmus an dieser Stelle mit $h_4 = 8$ ergibt sich somit die in Abbildung 1.3 dargestellte Reihenfolge der Schlüssel:

[8] hier wie im folgenden ohne Betrachtung der Schleifenbedingung, die an dieser Stelle insgesamt $lg(n)+1$-mal aufgerufen wird
[9] $8, 4, 2, 1$ für das gegebene Beispiel

n	**c2**	$O(n^{1.1})$	$O(n^{1.3})$	$O(n\ log\ n)$	$O(n^2)$
8	17	9	14	24	64
16	49	21	36	64	256
32	129	45	90	160	1024
64	321	97	222	384	4096
128	769	207	548	896	16.384
256	1793	445	1351	2048	65.536
1024	9217	2048	8192	10240	1.048.576
..					
2156	21.565	$4.645,29$	$21.564,69$	$23.875,84$	4.648.336
2157	21.576	$4.647,66$	$21.577,70$	$23.888,36$	4.652.649
2158	21.586	$4.650,03$	$21.590,70$	$23.900,88$	4.656.964
..					
10.000	120.005	25118	158.489	132.877	$1.0E8$
100000	1.500.006	316.227	3.162.277	1.660.964	$1.0E10$
200000	3.200.006	677.849	7.786.440	3.521.928	$4.0E10$
500.000	8.500.007	1.857.235	$2.56..E7$	9.465.784	$2.5E11$
1.000.000	18.000.007	3.981.071	$6.30..E7$	$1.99..E7$	$1.0E12$

Tabelle 1.1. Die Aufrufzahlen für $c2$ für verschiedene n. Schlüsselvergleiche spielen an dieser Stelle keine Rolle. Grün hinterlegte Werte sind kleiner als $c2$, rote sind größer. Mit $n > 2156$ wächst $O(n^{\frac{4}{3}})$ schneller als $c2$ und mit $n >= 982$ schneller als $O(n\ log\ n)$.

Abb. 1.3. Nach dem ersten Durchgang sind die Schlüssel in den Abständen $h_4 = 8$ sortiert, es ergeben sich zwei sortierte Teilfolgen der Länge 8

Die weiteren Durchgänge des Algorithmus sortieren das Feld entsprechend der Größe h: Es sind danach jeweils Schlüssel mit den Abständen 4 (Abbildung 1.4), 2 (Abbildung 1.5), und im letzten Schritt vollständig sortiert (Abbildung 1.6):

Abb. 1.4. Die Sortierung für $h_3 = 4$, es sind 4 Folgen, deren Schlüssel jeweils den Abstand 4 haben.

Abb. 1.5. Im vorletzten Sortierschritt sind 8 Folgen der Länge 2 sortiert.

Abb. 1.6. Der letzte Durchgang des Algorithmus vergleicht Schlüssel mit Distanzfolgen der Länge 1, also direkt benachbarte Schlüssel.

Für die Berechnung der Anzahl der Aufrufe von $c3$ stellt man fest, das in diesem Fall in jedem Schritt *immer* 2 Elemente, die eine Distanz von h_s aufweisen, falsch sortiert sind.

Der Algorithmus tauscht also in diesem Fall in jedem Durchgang alle Schlüssel untereinander aus, die er über `min < arr[j-delta]` miteinander vergleicht, was folglich die maximale Anzahl von Schlüsselvertauschungen in dieser vergleichsbasierten Implementierung für ein Feld der Größe n ergibt, nämlich $\frac{n}{2}$. Insgesamt finden dadurch für $c3$

$$lg(n) * \frac{n}{2} \tag{1.5}$$

Aufrufe statt.

Mit der Anzahl der berechneten Aufrufe $c1, c2, c3$ ergibt sich somit für die Laufzeit $T(n)$ für diesen Fall

$$lg(n) + n * lg(n) - n + 1 + lg(n) * \frac{n}{2} \tag{1.6}$$

und zusammengefasst

$$f(n) = \frac{3}{2} * n * lg(n) + lg(n) - n + 1 \tag{1.7}$$

was nach Einsetzen zu

$$lg(16) + 16 * lg(16) - 16 + 1 + lg(16) * \frac{16}{2} = 85 \tag{1.8}$$

Aufrufen für unser Beispiel führt.

Nachweis der Komplexitätsklasse

Um O zu ermitteln, werden nun alle Konstanten der Funktion 1.7 eliminiert, und der "dominante" Summand in Abhängigkeit von n betrachtet, der in diesem Fall $lg(n) * n$ ist.

Wir vermuten ein $N - log - N$-Wachstum[10] (vgl. [OW17a, 5]), und wollen nun zeigen, dass $T(n)$ in $O(n\ log\ n)$ liegt.
Hierfür müssen wir ein geeignetes c und ein n_0 finden, so dass gilt:

$$f \in O(n\ log\ n) :\leftrightarrow \exists n_0 \in \mathbb{N}, c \in \mathbb{R}, c > 0 : \forall n \geq n_0 : f(n) \leq c * n * lg(n) \tag{1.9}$$

(vgl. [GD18a, 11]).

Beweis. Zu zeigen ist

$$\frac{3}{2} * n * lg(n) + lg(n) - n + 1 \leq c * n * lg(n) \tag{1.10}$$

Wir wählen für $n_0 = 1$ und $c = \frac{3}{2}$, denn es gilt sicher $\forall n \geq n_0 : \frac{3}{2} * n * lg(n) \leq \frac{3}{2} * n * lg(n)$.

Ausserdem gilt stets $\forall n \in \mathbb{N} : lg(n) < n$, woraus $lg(n) - n < 0$ folgt, und damit auch $lg(n) - n + 1 \leq 0$.
Insgesamt gilt also

$$n_0 = 1, c = \frac{3}{2} : \forall n \geq n_0 : \frac{3}{2} * n * lg(n) + lg(n) - n + 1 \leq c * n * lg(n) \tag{1.11}$$

womit $f = O(n * log(n))$ gezeigt ist.□

[10] mit $O(log\ n)$ bzw $O(n\ log\ n)$ nehmen wir für die O-Notation wieder die in der Fachliteratur gebräuchlichere Schreibweise auf. Sowohl *Güting und Dieker* als auch *Ottmann und Widmayer* weisen in [GD18a, 15] bzw. [OW17a, 5] darauf hin, dass die Angabe der Basis keine Rolle spielt, da sich Logarithmen mit verschiedenen Basen ohnehin nur durch einen konstanten Faktor unterscheiden (es gilt $log_a(x) = \frac{log_b(x)}{log_b(a)}$).

Worst-Case-Analyse

Unter der Annahme, dass ein in umgekehrter Reihenfolge sortiertes Feld zu einer Laufzeit von $O(n^2)$ bei dem **Shellsort**-Algorithmus führt, konnten wir mit dem in dem vorherigen Abschnitt gewählten Parametern nur eine Laufzeit von $O(n\ log\ n)$ nachweisen.

Tatsächlich stellt der Anwendungsfall nicht den worst-case für den Algorithmus dar, da ja gerade diese Form von Sortierreihenfolge dem Algorithmus die Vorsortierung der h-Folgen ermöglicht:

> The idea underlying Shellsort is that moving elements of A long distances at each swap in the early stages, then shorter distances later, may reduce this $O(n^2)$ bound. ([Pra72, 3])

Es gilt also, die Vorsortierung auszuhebeln.

Für **Insertion-Sort** ist die Laufzeit im worst-case $O(n^2)$ (vgl. [OW17b, 87]). Da Shellsort mindestens im letzten Schritt diese Sortiermethoden auf Distanzfolgen der Länge 1 anwendet, muss der Algorithmus eine Folge als Eingabe erhalten, die durch die ersten $lg(n) - 1$-Durchgänge (mit $h_s = 2^{s-1}, 1 <= s < lg(n)$) keine Änderungen an der Schlüsselfolge vornimmt, um dann im allerletzten Schritt alle Daten zu sortieren, was maximal $\frac{n}{2}$ Inversionen bedeutet zuzüglich der benötigten Verschiebe-Operationen.
Hierzu kann wie im vorherigen Beispiel für $n = 16$ ein Feld mit folgender Schlüsselanordnung verwendet werden: Felder mit geradem Index enthalten kleinere Schlüssel als Felder mit ungeradem Index. Hier gilt für alle Elemente aus dem Feld A:

$$\forall i, j \in \mathbb{N}_{[0,n-1]}, 2 \mid i, 2 \nmid j : A[i] < A[j] \wedge A[i] < A[i+1] \wedge A[j] < A[j+1] \quad (1.12)$$

Abbildung 1.7 veranschaulicht die Anordnung.

Abb. 1.7. Eine *worst-case* Schlüsselfolge für Shellsort. Felder mit geradem Index enthalten kleinere Schlüssel als Felder mit ungeradem Index.

Markiert sind die direkt benachbarten Felder, die eine *Inversion* aufweisen[11], die im letzten Schritt des Algorithmus eine bzw. mehrere Verschiebungen bedingen.

[11] in diesem Zusammenhang bedeutet Inversion: *Fehlstellung* (vgl. [OW17b, 87])

In den vorherigen Schritten - also bei den Durchgängen mit Distanzfolgen $h_s > 1$, findet der Algorithmus jeweils Schlüssel in korrekter Sortierreihenfolge vor.

Mit $h_4 = 8$ werden die Felder $A[0..7]$ mit den Feldern $A[8..15]$ verglichen - hier gilt in jedem Fall, dass die Schlüssel $A[i] < A[j] (\forall i < j, j - i = 8)$ sind.

Auch im darauffolgenden Durchgang ($h_3 = 4, j - i = 4$) finden keine Verschiebungen statt, da keine Inversion gefunden wird. Erst im letzten Schritt, in dem direkt benachbarte Elemente miteinander verglichen werden, werden die Inversionen ermittelt und die Verschiebung der Elemente findet statt (s. Abbildung 1.8).

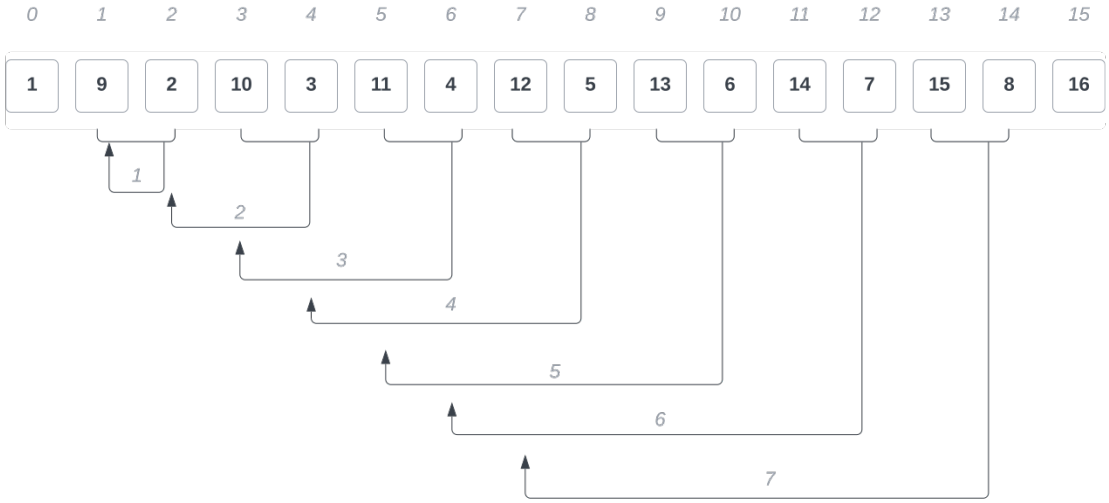

Abb. 1.8. Für die *worst-case* Schlüsselfolge werden im letzten Schritt 7 Fehlstellungen festgestellt. Jede Fehlstellung bedingt eine Verschiebung um die angegebene Anzahl von Positionen.

In diesem Fall wird $c3$ insgesamt 32 mal aufgerufen.
Da jeweils zwei Schlüssel bereits korrekt sortiert sind ($A[0]$ und $A[n-1]$) existieren noch $\frac{n}{2} - 1$ Inversionen. Jede Inversion erzwingt eine Verschiebung des größten Elements um $\frac{i}{2}$ Positionen nach links.
Für die Berechnung der Laufzeitkomplexität ergibt sich somit der Term

$$\sum_{i=1}^{\frac{n}{2}-1} i = \frac{\frac{n}{2}(\frac{n}{2} - 1)}{2} = \frac{n^2 - 2n}{8} \tag{1.13}$$

und unter Berücksichtigung der Terme für $c1$, $c2$, $c3$ die Funktion

$$f(n) = lg(n) + n * lg(n) - n + 1 + \frac{n^2 - 2n}{8} \tag{1.14}$$

was zu einer Laufzeitabschätzung von $O(n^2)$ führt.

2

Bubblesort

Lösung

- Benachbarte Elemente werden vertauscht, wenn sie in der falschen Reihenfolge sind.
- Der Algorithmus ist stabil.

Anmerkungen und Ergänzungen

Beide Aussagen finden sich im Skript (Teil 2) auf Seite 165, auf Seite 166 außerdem ein Beispiel, das das Vertauschen direkt benachbarter Elemente demonstriert. Etwas formaler geben *Ottmann und Widmayer* in [OW17b, 89 ff.] eine Laufzeitanalyse an.

Mit *"Direkte Auswahl"* bezeichnet man auch **Selection-Sort**[1].

Bubblesort gehört zu den *allgemeinen Sortierverfahren* und besitzt deshalb eine untere Schranke von $\Omega(n\ log\ n)$[2]. Aus diesem Grund werden auch kleinere unsortierte Mengen nicht in $O(n)$ sortiert[3], es findet außerdem keine sequentielle Suche nach dem kleinsten Element statt.

[1] s. Skript (Teil 2) S. 159
[2] [OW17b, 154]
[3] s. Skript (Teil 2) S.179 ff., außerdem [OW17b, 153 ff.].

3

Effizienz des Quicksort-Algorithmus

Lösung

- von den gewählten Vergleichswerten
- von der Rekursionstiefe

Anmerkungen und Ergänzungen

Die Vergleichswerte können durch die Implementierung gewählt werden, sie bedingen die Schrittfolgen. Da die Schrittfolgen nicht gewählt werden können, fällt die Option als richtige Antwort raus.
Die Rekursionstiefe bedingt die Effizienz eines Algorithmus: Je tiefer diese ist, desto mehr Operationen werden durchgeführt.

Wichtige Maße für die Effizienz eines Algorithmus sind der benötigte Speicherplatz sowie die benötigte Rechenzeit zur Ausführung (vgl. [OW17a, 2], außerdem Skript (Teil 2) S. 157).
Das Skript (Teil 2) stellt auf Seite 177 fest, dass der Aufwand für das Sortieren mit Quicksort wesentlich dadurch bestimmt wird, wie die Folgen in Teilfolgen aufgeteilt werden: So sind bspw. für Folgen der Länge n Teilfolgen der Längen 1 und $n-1$ möglich, wodurch ein Aufrufbaum *entarten* kann und hohe Rekursionstiefe entsteht (vgl. [GD18b, 177 f.])

Ottmann und Widmayer stellen fest:

> Die im ungünstigsten Fall auszuführende Anzahl von Schlüsselvergleichen und Bewegungen hängt damit stark von der Anzahl der Aufteilungsschritte und damit von der Zahl der initiierten rekursiven Aufrufe ab. ([OW17b, 96])

Sie stellen zur Verbesserung der Effizienz die 3-Median-Strategie vor (vgl. [OW17b, 102] sowie [GD18b, 183]), die auch im Skript (Teil 2) auf Seite 177 zur Auswahl des Pivotelements gezeigt wird.

4

McCarthy-Funktion

Lösung

- für $1 \leq i \leq 100$ liefert die Funktion den Wert $f(i) = 91$
- für $n > 100$ liefert die Funktion den Wert $f(n) = (n - 10)$
- die maximale Rekursionstiefe von 19 tritt bei $n = 1$ auf

Anmerkungen und Ergänzungen

Die *Rekursionstiefe* ist i.A. nicht mit der Anzahl der Methodenaufrufe gleichzusetzen, worauf das Skript (Teil 2) auf Seite 34 hinweist.

Die **Tiefe** eines Knotens ist im Skript auf Seite 101 beschrieben, für sie gilt

$$T(n) = \begin{cases} 0 \; falls \; n \; Wurzel \\ 1 \; + \; T(n') \; sonst \; (mit \; n' \; = \; Elterknoten) \end{cases} \tag{4.1}$$

Die maximale Rekursionstiefe kann für die `McCarthy 91`-Methode programmatisch wie folgt ermittelt werden (s. Listing 4.1):

```
1
2  private static m91depth = 0;
3
4  public static int m(int n) {
5      return f(n, 0);
6  }
7
8  private static int m(int n, int depth) {
9
10     m91depth = Math.max(m91depth, depth);
11
12     if (n > 100) {
13         return n - 10;
14     }
15
16     return m(
17         m(n + 11, depth + 1),
18         depth + 1
19     );
20
```

```
21     }
```

Listing 4.1. Ermittlung der Rekursiontiefe für die McCarthy 91-Funktion. Die Rekursionstiefe wird in einer statischen Variable gespeichert.

Abbildung 4.1 zeigt den Aufrufbaum für $n = 97$:

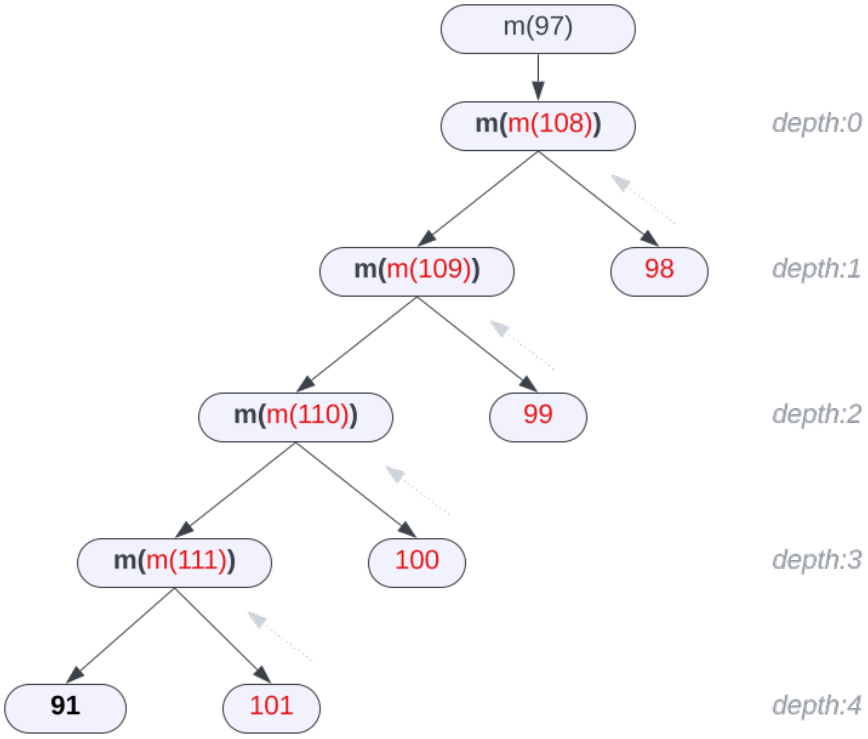

Abb. 4.1. Der Aufrufbaum für $m(97)$: Rechte Knoten werden als Rückgabewert der inneren Aufrufe an die äußeren Methoden der Vorgängerknoten übergeben, aus denen die Methodenaufrufe in den linken Teilbäumen resultieren.

5

Sortieralgorithmen (Klassifikation)

Lösung

- Stabile Verfahren bewahren die Reihenfolge der Datensätze, falls deren Sortierschlüssel gleich sind.
- Bei internen Verfahren werden alle Datensätze im Hauptspeicher gehalten.

Anmerkungen und Ergänzungen

Externe Verfahren sortieren Datensätze auf Externspeichern wie bspw. Magnetbänder (vgl. [OW17b, 141 ff.]). Hierfür müssen *extern* zu sortierende Daten in Teilen in den Hauptspeicher geladen werden, im Gegensatz zu rein *internen* Verfahren, die alle Datensätze im Hauptspeicher halten.

in situ bzw. **in place** Verfahren sortieren Schlüssel in einem Feld alleine durch Vertauschen *innerhalb* des Feldes (vgl. [GD18b, 169]), ohne zusätzlich angelegte Felder zu nutzen. Ein Beispiel für eine *in situ*-Sortierung ist Bubble Sort, bei dem direkt benachbarte Elemente getauscht werden, wenn sie in der falschen Reihenfolge zueinander stehen. Bei **Merge Sort** werden hingegen in Teilschritten Felder angelegt, die jeweils einen Ausschnitt der zu sortierenden Daten beinhalten (vgl. [OW17b, 112 ff.]).

Mit *Merge Sort* als Beispiel kann auch die Aussage widerlegt werden, dass *in place*-Verfahren nur genau den Hauptspeicher belegen, der von den zu sortierenden Elementen schon belegt ist: Wenn ein Feld der Länge n gegeben ist, werden $2 * n - 2$ zusätzliche Felder in den Teilschritten angelegt - es wird also zusätzlicher Speicher angefordert.

Per Defintion bewahren **stabile Verfahren** die relative Reihenfolge der Daten bei (vgl. [GD18b, 170]). Ein Beispiel für ein nicht-stabiles Verfahren ist **Selection Sort**. Im folgenden werden die Schlüssel $1, 4_1, 5, 4_2, 2$ sortiert. Da die 4 mehrfach vorkommt, ist sie entsprechender der ursprünglichen Reihenfolge nummeriert. Abbildung 5.1 stellt die Sortierung dar. Im letzten Schritt haben 4_1 und 4_2 ihre Reihenfolge getauscht.

Abb. 5.1. Ein Beispiel für eine instabile Sortierung (Selection Sort): Nach Anwendung des Verfahrens ist die Reihenfolge von 4_1 und 4_2 vertauscht.

6

Sortieralgorithmen (Komplexität)

Lösung

- Alle auf Vergleichen beruhenden Sortierverfahren sind in der Laufzeitklasse $\Omega(n\ log\ n)$
- MergeSort ist in der Laufzeitklasse (n^2)
- MergeSort ist in der Laufzeitklasse $(n\ log\ n)$
- QuickSort ist in der Laufzeitklasse (n^2)

Anmerkungen und Ergänzungen

> Jedes allgemeine Sortierverfahren benötigt zum Sortieren von N verschiedenen Schlüsseln sowohl im schlechtesten Fall als auch im Mittel wenigstens $\Omega(N\ log\ N)$ Schlüsselvergleiche. ([OW17b, 154, Satz 2.4])

Aus diesem Grund gilt die Aussage, dass alle auf Vergleichen beruhenden Sortierverfahren in der Laufzeitklasse $\Omega(n\ log\ n)$ sind.

Für *Merge Sort* gilt auch im *worst-case* die obere Schranke $O(n\ log\ n)$ (vgl.([OW17b, 116])).

QuickSort benötigt im Mittel eine Laufzeit von $O(n\ log\ n)$ (vgl. [OW17b, 99]) sowie im worst-case quadratische Laufzeit (vgl. [OW17b, 96], außerdem Abschnitt 3).

7

Wahl des Algorithmus

Lösung

- Direktes Einfügen

Anmerkungen und Ergänzungen

Von den angegebenen Optionen (Bubble-, Selection-, und Insertion Sort) eignet sich **Insertion Sort** am ehesten, wenn ein Großteil des Feldes bereits vorsortiert ist, und nur wenige Elemente an der falschen Position stehen (s. Skript (Teil 2) S. 164).

Ottmann und Widmayer führen für vorsortierte Felder **Smoothsort** an (vgl. [OW17b, 112]), eine von *Dijkstra* entwickelte Variante von **Heap Sort**[1], die im *best case* $O(n)$ Zeit benötigt (vgl. [Dij81]).

[1] s. `https://en.wikipedia.org/wiki/Heapsort`

8

wastueIch

Lösung

- Es handelt sich um das Verfahren der “binären Suche“.
- Die Zeitkomplexität des Algorithmus liegt bei $O(log_2\ n)$.
- Die Methode arbeitet rekursiv

Anmerkungen und Ergänzungen

Eine Beschreibung des Verfahrens findet sich bei *Ottmann und Widmayer* ([OW17c, 174]) sowie Güting und Dieker ([GD18a, 17]).

Literaturverzeichnis

[CL22] Thomas H Cormen und Charles E Leiserson. *Introduction to Algorithms, fourth edition*. en. London, England: MIT Press, Apr. 2022. ISBN: 9780262046305.

[Dij81] Edsger W. Dijkstra. „Smoothsort, an alternative for sorting in situ“. circulated privately. Aug. 1981. URL: http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD796a.PDF.

[GD18a] Ralf Hartmut Güting und Stefan Dieker. „Einführung“. In: *Datenstrukturen und Algorithmen*. Wiesbaden: Springer Fachmedien Wiesbaden, 2018, S. 1–38. ISBN: 978-3-658-04676-7. DOI: 10.1007/978-3-658-04676-7_1. URL: https://doi.org/10.1007/978-3-658-04676-7_1.

[GD18b] Ralf Hartmut Güting und Stefan Dieker. „Sortieralgorithmen“. In: *Datenstrukturen und Algorithmen*. Wiesbaden: Springer Fachmedien Wiesbaden, 2018, S. 169–200. ISBN: 978-3-658-04676-7. DOI: 10.1007/978-3-658-04676-7_5. URL: https://doi.org/10.1007/978-3-658-04676-7_5.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. USA: Addison Wesley Longman Publishing Co., Inc., 1997. ISBN: 0201896834.

[OW17a] Thomas Ottmann und Peter Widmayer. „Grundlagen“. In: *Algorithmen und Datenstrukturen*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, S. 1–78. ISBN: 978-3-662-55650-4. DOI: 10.1007/978-3-662-55650-4_1. URL: https://doi.org/10.1007/978-3-662-55650-4_1.

[OW17b] Thomas Ottmann und Peter Widmayer. „Sortieren“. In: *Algorithmen und Datenstrukturen*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, S. 79–165. ISBN: 978-3-662-55650-4. DOI: 10.1007/978-3-662-55650-4_2. URL: https://doi.org/10.1007/978-3-662-55650-4_2.

[OW17c] Thomas Ottmann und Peter Widmayer. „Suchen“. In: *Algorithmen und Datenstrukturen*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, S. 167–189. ISBN: 978-3-662-55650-4. DOI: 10.1007/978-3-662-55650-4_3. URL: https://doi.org/10.1007/978-3-662-55650-4_3.

[Pra72] Vaughan R. Pratt. „Shellsort and Sorting Networks“. In: *Outstanding Dissertations in the Computer Sciences*. 1972. URL: https://api.semanticscholar.org/CorpusID:11928873.

[PS65] AA Papernov und GV Stasevich. „A method of information sorting in computer memories“. In: *Problemy Peredachi Informatsii* 1.3 (1965), S. 81–98.

[She59] D. L. Shell. „A High-Speed Sorting Procedure“. In: *Commun. ACM* 2.7 (Juli 1959), S. 30–32. ISSN: 0001-0782. DOI: 10.1145/368370.368387. URL: https://doi.org/10.1145/368370.368387.