Technische Universität
Kaiserslautern

Fachbereich Informatik
AG Algorithmen und Komplexität

Ausgabedatum: 16.01.2014
Version: 2014-01-15 19:15

12. Übungsblatt zur Vorlesung Entwurf und Analyse von Algorithmen, WS 13/14

Abgabe: *Bis* ***Freitag****, 24.01.2014, 12:00 Uhr, Kasten im Treppenhaus 48-6.*

Bitte denken Sie an die Anmeldung zur 2. Zwischenklausur (siehe OLAT/HP) und an die Vorlesungsumfrage (Tokens im SCI erhältlich).

Der Fehler der Woche

Quicksort tauscht nur, siehe also Analyse von Bubblesort.

Was ist falsch?

Notationshinweis

Wir schreiben $\mathcal{B}(1,p)$ für die Bernoulli-Verteilung mit Parameter p, $\mathcal{U}[a,b)$ für die Uniformverteilung über $[a,b) \subseteq \mathbb{R}$ und $\mathcal{U}[1..n]$ für die Uniformverteilung über $\{1,\ldots,n\}$.

Basisaufgaben

B1: Speicherbedarf von Sortieralgorithmen

2 Punkte

Welche Sortieralgorithmen, die Sie aus der Vorlesung kennen, brauchen wie viel Speicher (zusätzlich zur Eingabe)?

Geben Sie für die Algorithmen

- Bubblesort,
- Shellsort,
- Quicksort und
- 2-Wege-Mergesort

jeweils an, in welchen der Komplexitätsklassen $\mathcal{O}(1)$, $\mathcal{O}(\log n)$, $\mathcal{O}(n)$ und $\omega(n)$ der (zusätzliche) Speicherbedarf schlimmstenfalls liegt. Hierbei ist n die Länge des Eingabearrays.

B2: Stabilität

3 Punkte

In der Praxis kommt es oft vor, dass mehrere unterschiedliche Einträge einer Wörterbuchdatenstruktur den gleichen Schlüssel haben. Ein Sortierverfahren, das die Reihenfolge der Einträge mit gleichem Schlüssel untereinander nicht ändert, heißt *stabil*. Dies ist zum Beispiel wünschenswert, wenn man sukzessive nach mehreren Schlüsseln sortieren möchte.

Untersuchen Sie die Sortieralgorithmen aus der Vorlesung auf Stabilität! Geben Sie jeweils eine präzise Begründung für Ihre Behauptung an.

Geben Sie für die nicht stabilen Verfahren außerdem an, ob und wie man sie mit kleinen Änderungen stabil machen kann.

a) Insertion-Sort

b) Quicksort

c) Mergesort

Hinweis: Untersuchen Sie genau die im Buch gegebenen Implementierungen!

B3: Inversionen zählen

3 Punkte

Erweitern Sie Mergesort so, dass der Algorithmus während des Sortierens auch die Anzahl der beseitigten Inversionen berechnet und zurückgibt. Dabei soll die asymptotische Laufzeit nicht verschlechtert werden.

Begründen Sie die Korrektheit und Effizienz Ihrer Erweiterung.

B4: Algorithmenanalyse

4 Punkte

Betrachten Sie folgenden Algorithmus `Sort`, der auf ein als global definiertes Feld `A` des Typs `ARRAY[1..n] OF CARDINAL` angewendet wird:

```
1  Sort(i,j : CARDINAL);
2    VAR c,k : CARDINAL;
3  BEGIN
4    IF ( i + 1 < j ) THEN
5      k := FLOOR((j-i+1)/3);
6      Sort(i, j-k);
7      Sort(i+k, j);
8      Sort(i, j-k);
9    ELSE
10     IF ( A[i] > A[j] ) THEN
11       c := A[i];
12       A[i] := A[j];
13       A[j] := c;
14     END;
15   END;
16 END;
```

Wie viele Vergleiche werden für die Abarbeitung des Aufrufs `Sort(1,n)` im schlechtesten Fall durchgeführt? Geben Sie die entsprechende Θ-Klasse an.

Beweisen oder widerlegen Sie außerdem, dass der Aufruf `Sort(1,n)` das Feld `A` sortiert.

Aufbauaufgaben

37. Aufgabe

$1 + 2 + 1$ Punkte

Ein *Shuffle-Algorithmus* ordnet Eingabearrays beliebiger Größe $n \in \mathbb{N}$ so um, dass jede Permutation der Eingabe als Ergebnis möglich und gleich wahrscheinlich ist.

a) Zeigen Sie, dass nicht für jedes $n \in \mathbb{N}$ unter Verwendung von je endlich vielen $\mathcal{B}(1, 1/2)$-verteilten Bits (und keinen anderen Zufallsquellen) $\mathcal{U}[1..n]$-verteilte Zahlen gezogen werden können.

b) Entwerfen Sie einen Shuffle-Algorithmus, der mit einer $\mathcal{U}[0, 1)$-Quelle im Worst-Case in Zeit $\mathcal{O}(n)$ läuft, und analysieren Sie die nötige Anzahl Zufallszahlen. Begründen Sie Ihre Behauptungen.

c) Entwerfen Sie einen Shuffle-Algorithmus, der nur eine $\mathcal{B}(1, 1/2)$-Quelle verwendet. Begründen Sie die Korrektheit und analysieren Sie sowohl die Anzahl verwendeter Zufallsbits als auch die Gesamtlaufzeit im Worst- und Average-Case.

38. Aufgabe

$1 + 1 + 2$ Punkte

Untersuchen Sie die Sortieralgorithmen aus der Vorlesung auf Stabilität! Geben Sie jeweils eine präzise Begründung für Ihre Behauptung an.

Geben Sie für die nicht stabilen Verfahren außerdem an, ob und wie man sie mit kleinen Änderungen stabil machen kann.

a) Shell-Sort

b) Heapsort

c) Angenommen, Sie müssen einen nicht stabilen Sortieralgorithmus A für ganze Zahlen verwenden, brauchen aber Stabilität. Können Sie die Eingabe so transformieren, dass das sortierte Ergebnis stabil ist, ohne die asymptotische Laufzeit von A zu verschlechtern? Sie können dabei annehmen, dass Sie A neben der Eingabefolge eine Vergleichsfunktion übergeben können.

   Entwerfen Sie einen entsprechenden Algorithmus, begründen Sie seine Korrektheit und analysieren Sie die Laufzeit.

Hinweis: Untersuchen Sie genau die im Buch gegebenen Implementierungen!

39. Aufgabe

$2 + 2 + 1$ Punkte

In der Vorlesung haben wir Quicksort unter der Annahme analysiert, die Eingabe sei eine uniform zufällig gewählte Permutation. In der Praxis wird diese Annahme in den wenigsten Fällen erfüllt sein; wie sieht es mit anderen Eingabeverteilungen aus?

Wie verhält sich die Θ-Asymptotik der erwarteten Laufzeit von Quicksort (Implementierung der Vorlesung) auf den folgenden Eingaben? Analysieren Sie die jeweiligen Extremfälle und diskutieren Sie, inwiefern etwaige negative Effekte auch für weniger extreme Fälle auftreten.

Schlagen Sie – sofern nötig – Verbesserungen an unserer Implementierung vor, die den beobachteten Effekten entgegenwirken.

a) Duplikate – Schlüssel können mehrfach vorkommen.

b) Vorsortierung – die Eingaben sind schon teilweise sortiert, haben also sehr wenig Inversionen.

c) Inverse Vorsortierung – die Eingaben sind teilweise vorsortiert, aber *falsch herum*, haben also viele Inversionen.

40. Aufgabe

$2 + 1 + 1 + 2$ Punkte

Betrachten Sie den folgenden Algorithmus:

```
1   procedure sort(A) {
2     n = A.length
3     B = new Array[0..n-1]
4
5     for ( i = 0 to n-1 ) {
6       B[i] = new List
7     }
8
9     for ( i = 0 to n-1 ) {
10      B[floor(n * A[i])].append(A[i])
11    }
12
13    for ( i = 0 to n-1 ) {
14      B[i] = insertionSort(B[i])
15    }
16
17    j = 0
18    for ( i = 0 to n-1 ) {
19      if ( B[i].length > 0 ) {
20        cur = B[i].first
21        while ( cur != NIL ) {
22          A[j] = cur.value
23          cur = cur.next
24          j = j + 1
25        }
26      }
27    }
28  }
```

Die Prozedur `floor` gibt hier die größte ganze Zahl zurück, die kleiner oder gleich dem Parameter ist. `insertionSort` arbeitet analog zu „Sortieren durch direktes Einfügen“ aus der Vorlesung, nur auf Listen statt in Arrays.

a) Zeigen Sie, dass `sort(A)` das Array `A` sortiert, wenn dieses Zahlen aus $[0, 1)$ enthält.

b) Geben Sie eine Best-Case-Eingabe für `sort` (allgemein in Länge des Arrays n) an und leiten Sie die Best-Case-Laufzeit als Θ-Klasse in n ab.

c) Geben Sie eine Worst-Case-Eingabe für `sort` (allgemein in n) an und leiten Sie die Worst-Case-Laufzeit als Θ-Klasse in n ab.

d) Nehmen Sie an, dass die Einträge von `A` uniform zufällig aus $[0, 1)$ gezogen wurden. Bestimmen Sie die Θ-Klasse (in n) der erwarteten Laufzeit von `sort(A)` und beweisen Sie Ihre Behauptung.