Prof. Dr. Stefan Göller
Dr. Da-Jung Cho

Einführung in die Informatik

WS 2019/2020

Übungsblatt 11
24.01.2020 - 30.01.2020

Abgabe: Bis zum 30.01.20 18:00 Uhr über moodle. **Beachten Sie bitte folgende Hinweise zur Abgabe.** Verwenden Sie keine globale Variablen, löschen Sie alle Ihre `print`-Anweisungen (außer natürlich diejenigen, die wir selbst in den Templates bereitstellen!) und fügen Sie möglichen Testcode in den Bereich ein, den wir in den Templates dafür vorgeben, nämlich nach `if __name__ == "__main__"`. Wenn diese obigen Vorgaben nicht eingehalten werden, werden Punkte abgezogen.

Aufgabe 1 (Sortieren durch Suchen) (20 Punkte):

Schreiben Sie zunächst eine Funktion `einfuegen(L,e)`, die eine *aufsteigend sortierte Liste* [1] `L` und eine Position $p \in \{0, \ldots, \mathbf{len}(\texttt{L})\}$ zurückgibt, an welche das Element `e` eingefügt werden kann: D.h. alle Elemente in `L` an Position $\geq p$ sind größer oder gleich `e` und alle Elemente an Position $\leq p$ sind kleiner oder gleich `e`; falls `L` leer ist soll $p = 0$ zurückgegeben werden. Ihre Funktion soll eine leichte Anpassung von Bisection Search sein und eine Laufzeit haben, die logarithmisch in der Länge von `L` ist.

Schreiben Sie dann eine Funktion `sortiere(L)`, die eine Liste `L` bestehend aus n Integern entgegennimmt und diese in $O(n \log n)$ sortiert. Diese Funktion soll startend auf einer leeren Liste wiederholt alle Elemente von `L` von links nach rechts durchgehen und mittels `einfuegen` eine einzufügende Position bestimmen. Fügen Sie das jeweils aktuelle Element durch die Methode `insert` in Ihre sortierte Liste ein (Sie dürfen annehmen, dass `insert` in konstanter Zeit läuft). Ihre Funktion `sortiere` darf maximal 10 Zeilen besitzen.

[1] Sie dürfen hierzu also annehmen, dass die übergebene Liste `L` bereits aufsteigend sortiert vorliegt.

Aufgabe 2 (Quicksort) (5*5=25 Punkte):

Quicksort gilt als einer der schnellsten Sortieralgorithmen in der Praxis. Wie in der Vorlesung gezeigt, gibt es allerdings Eingaben (z.B. umgekehrt sortierte Listen), auf denen Quicksort sehr schlechte Laufzeit hat. Es existieren Varianten, die derartiges Verhalten eindämmen sollen, ohne die im Allgemeinen gute Laufzeit von Quicksort zu verlieren. Zu den bekannteren gehören:

i) (Introsort) Diese Variante von Quicksort startet mit einer Schranke für die Tiefe rekursiver Aufrufe (nämlich $2 \cdot \log(n)$, wobei n die Länge der Eingabe ist). Mit jedem rekursiven Aufruf wird die Schranke um eins verringert. Sobald die Schranke auf 0 fällt, gibt Quicksort auf und sortiert die Eingabe, für die das Limit auf 0 gefallen ist, mit einem Verfahren, welches garantierte Laufzeit in $O(n \log(n))$ hat, nämlich Mergesort.

ii) (Sedgewick-Cutoff): Es fällt auf, dass Quicksort auf sehr kleinen Eingaben solchen Algorithmen unterlegen ist, welche weniger aufwändig konzipiert sind. Ein verbreiteter Ansatz ist es, Eingaben der Länge kleiner als ein gegebenes n (für uns 9) an beispielsweise Insertionsort abzugeben. Diese Quicksort-Variante testet, ob Ihre Eingabelänge weniger als 9 ist. Falls ja, wird die Eingabe mittels Insertion Sort sortiert, falls nein, wird die Eingabe nach dem Quicksort-Verfahren geteilt und rekursiv weiter verfahren. Sobald die rekursiven Aufrufe Länge weniger als 9 erreichen, wird dann an Insertionsort übergeben.

iii) (Median-aus-3-Pivotwahl) Für diese Variante wird nicht immer das erste Element als Pivot gewählt, sondern es werden das erste Element der Liste, das letzte Element der Liste sowie das Element in der Mitte (also an Position $n//2$) verglichen. Das zweitgrößte dieser drei Elemente wird dann als Pivot verwendet. Bei Listen der Länge weniger als 3 wird das erste Element als Pivot gewählt.

iv) (Median-aus-3 mit Cutoff) Für diese Variante werden Eingaben der Länge kleiner 9 wie oben beschrieben mittels Insertionsort sortiert, während für größere Eingaben das Pivot-Element nach dem Median-aus-3-Verfahren gewählt wird.

v) (Zufälliges Pivotelement) Für diese Variante wird das Pivotelement gleichverteilt zufällig gewählt[2]. Dabei macht man sich zunutze, dass es dabei sehr unwahrscheinlich ist, konsequent das größte bzw. das kleinste Element zu wählen.

Setzen sie sich nun mit den Programmen in der Rumpfdatei auseinander und verstehen Sie, wie sie funktionieren. Beachten Sie, dass die Funktionen jeweils die Anzahl der von ihnen angestellten Vergleiche von Listenelementen protokollieren.

[2]Verwenden Sie dazu das Paket `random` und die Funktion `random.choice()`, um einen Index in $0, \ldots, n-1$ zu finden, wobei n die Listenlänge ist.

a) Ergänzen Sie jeweils die Funktionsköpfe `qsort_intro(liste)`, `qsort_cutoff(liste)`, `qsort_ma3(liste)`, `qsort_cutoff_ma3(liste)` und `qsort_random(liste)` so zu einer vollen Funktion, dass diese Funktionen sich jeweils wie die oben beschriebenen Quicksort-Varianten verhalten. Beachten Sie, dass Sie genau wie in den vorgegebenen Funktionen die Anzahl der von Ihnen anstellten Vergleiche von Listenelementen protokollieren sollen.

   Hinweis: Möglicherweise benötigen Sie Hilfsfunktionen für einige Funktionen. Verwenden Sie keine weiteren globalen Variablen.

b) (Sternchenaufgabe) Testen Sie nun die folgenden Verfahren gegeneinander: reguläres Quicksort, Insertionsort, Mergesort und alle fünf von Ihnen programmierten Quicksort-Varianten. Verwenden Sie die vorgebene Schablone, um sowohl Laufzeit als auch die Anzahl der benötigten Vergleiche zu ermitteln. Als Eingabe verwenden wir jeweils je eine aufsteigend und absteigend sortierte Liste, sowie je 5 zufällig erstellte Listen. Erläutern Sie, welche Algorithmen jeweils auf welcher Sorte von Eingaben am besten funktionieren.

Aufgabe 3 (Radixsort) (10+5 Punkte):

In dieser Aufgabe werden wir ein Sortierverfahren behandeln, welches sowohl schneller läuft als Quicksort als auch als Mergesort, wenn man sich die Struktur der zu sortierenden Daten zunutze macht. Ein klassisches Beispiel für so einen Anwendungsfall ist, wenn ein Datensatz nach lexikographischer Ordnung sortiert werden soll, beispielsweise Matrikelnummern.

In so einem Fall bietet es sich an, *Radixsort* zu verwenden. Radixsort funktioniert folgendermaßen: Gegeben eine Liste nichtnegativer Integer, prüft Radixsort, ob die Länge der Liste höchstens eins ist. In diesem Fall ist die Liste schon sortiert und wird direkt zurück gegeben. Ansonsten bildet Radixsort zunächst zehn Unterlisten, nämlich die Liste der nichtnegativen Integer, die mit einer 0 anfangen, die Liste derer, die mit einer 1 anfangen, usw. Diese zehn Listen können in $O(n)$ gebildet werden. Danach werden diese einzelnen Listen rekursiv mittels Radixsort sortiert (natürlich wird dann die zweite Stelle der Zahlen betrachtet, danach die dritte, usw.), und die sortierten Listen werden dann in der richtigen Reihenfolge zusammengesetzt. Listen von n nichtnegativen Integern, die jeweils die Stelligkeit s haben, können so in $O(n \cdot s)$ sortiert werden.

Allerdings funktioniert Radixsort für sich alleine genommen nicht besonders gut. Das liegt daran, dass sehr viel Zeit für die letzten Schritte aufgewendet werden muss, in denen sehr kleine Listen bearbeitet werden. Radixsort eignet sich aber hervorragend dazu, eine Liste zunächst grob vorzusortieren. Dafür wird nur bis zu einer bestimmten Stelligkeit vorsortiert. Die Gesamtliste, die nun vorsortiert, aber nicht unbedingt sortiert ist (siehe Beispiel unten),

kann dann mit Bubblesort endgültig sortiert werden. Bubblesort hat zwar im Allgemeinen eine recht schlechte Laufzeit, es profitiert aber sehr davon, wenn eine Liste schon annähernd sortiert ist. In Kombination können Radixsort und Bubblesort auf passenden Eingaben selbst einen so mächtigen Algorithmus wie Quicksort spielend schlagen.

Wir wollen jetzt den ersten Teil dieses Verfahrens implementieren, nämlich Radixsort. Gesucht ist ein Verfahren, welches, gegeben eine Zahl s und eine Liste nichtnegativer Integer, diese Liste so vorsortiert, dass die Liste sortiert ist, wenn man nur die ersten s Stellen jedes Integers betrachtet. Wir zählen Stellen ab Position 0.

a) Realisieren Sie den Funktionskopf `radixsort(liste, s)` zu einer Funktion, die Radixsort wie oben beschrieben bis zur Tiefe `s` durchführt. Dabei ist `liste` die zu sortierende Liste und `s` gibt an, wann abgebrochen werden soll (dieser Wert soll bei rekursivem Aufruf entsprechend um eins verringert werden). Sobald `s` gleich 0 ist wird die Liste `liste` einfach unsortiert zurück gegeben. Der Parameter `s` gibt an, nach der wievieltletzten Stelle der Einträge in der Eingabeliste sortiert werden soll. Ist `s` also beispielsweise 1, dann sollen die Zahlen in `liste` so sortiert werden, dass in der Rückgabe alle Zahlen, deren erste Ziffer eine 0 ist, vor allen Zahlen kommen, deren erste Ziffer eine 1 ist, usw.

   Implementieren Sie Ihren Algorithmus so, dass er auf Eingabe `radixsort(liste, s)` in $O(n \cdot \texttt{s})$ läuft, wobei n die Länge von `liste` ist.

b) Benutzen sie die vorgebene Schablone in der Rumpfdatei, um Beispiele zu finden, in denen die oben beschriebene Kombination aus Radixsort und Bubblesort regulärem Quicksort und Mergesort überlegen ist. Die Schablone hat Parameter s, p und t. Dabei wird zunächst eine Liste der Länge 10^p erzeugt, die als Einträge t-stellige Zahlen enthält. Die Schablone sortiert diese Liste dann einmal mit Radixsort und Bubblesort, wobei die ersten s Stellen der Eingabeliste mit Radixsort vorsortiert werden (wir setzen also $s \leq t$ voraus), und dann Bubblesort übernimmt. Danach wird die ursprüngliche Liste noch jeweils einmal mit Mergesort und mit Quicksort sortiert. Die Laufzeiten der drei Verfahren werden dann ausgegeben.

   Finden Sie drei Kombinationen von s, p und t, so dass Radixsort und Bubblesort zusammen mindestens 10% schneller laufen als sowohl Mergesort als auch Quicksort.

Beispiel: Wir betrachten noch einen Lauf von Radixsort bis zur Tiefe $s = 1$ auf der Liste `[899, 747, 740, 650, 537, 737, 667]`. Die Liste hat Länge 7 und muss daher sortiert werden. Radixsort zerlegt die Eingabe nun in Teillisten für jede Ziffer. Die Listen der Einträge, die mit den Ziffern $0, 1, 2, 3, 4$ und 9 anfangen sind leer. Die Liste für die Ziffer 5 ist `[537]`, die Liste für die Ziffern 6 ist `[650, 667]`, die Liste für Ziffer 7 ist `[747, 740, 737]` und die Liste für die Ziffer 8 ist `[899]`.

Diese Listen werden jetzt rekursiv mit Radixsort sortiert, wobei es jetzt auf die nächste Stelle ankommt. Die Listen für die Ziffern 0 bis 5 haben Länge höchstens 1, sind daher bereits sortiert und werden direkt zurückgegeben. Die Liste für die Ziffer 6 enthält zwei Einträge. Radixsort legt also wieder zehn Listen an und verteilt die Einträge in der Liste `[650, 667]` auf diese zehn Listen. Die Listen für die Ziffern 0 bis 4 sowie 7 bis 9 sind leer, die Liste für die Ziffer 5 ist `[650]` und die Liste für die Ziffer 6 ist `[667]`. Da die entstandenen Listen alle Länge höchstens 1 haben, gibt Radixsort jetzt die Listen konkateniert in der richtigen Reihenfolge zurück, also die Liste `[650, 667]`. Als nächstes ist die Ziffer 7 an der Reihe. Die Aufteilung auf die einzelnen Ziffern ergibt hier leere Listen für die Ziffern $0, 1, 2$ sowie 5 bis 9. Die Liste für die Ziffer 3 ist `[737]`, die Liste für die Ziffer 4 ist `[747, 740]`. Da nur bis Stelligkeit 1 sortiert werden soll, gibt Radixsort diese Listen konkateniert als `[737, 747, 740]` zurück. Die Liste für die Ziffer 8 bezüglich Stelligkeit 0 ist eine Einerliste und bereits sortiert. Der Rückgabewert von Radixsort ist also `[537, 650, 667, 737, 747, 740, 899]`. Beachten Sie, dass diese Liste nicht vollständig sortiert ist.