TH Brandenburg
Online Studiengang Medieninformatik
Fachbereich Informatik und Medien
Algorithmen und Datenstrukturen
Prof. Dr. rer. nat. Ulrich Baum

Einsendeaufgabe 2

Sommersemester 2021

Abgabetermin 23.05.2021

Maximilian Schulke
Matrikel-Nr. 20215853

1 Finde die falsche Münze

a) Man kann das Problem lösen, indem man die erste Münze nacheinander mit allen anderen vergleicht. Geben Sie den Worst-Case und Best-Case Aufwand dafür an.

Falls der Best-Case entritt (die falsche Münze liegt an einer der ersten beiden Positionen) haben wir nur einen einzigen Vergleich dementsprechen $O(1)$. Der Worst- Case tritt dementsprechen im umgekehrten Fall ein, also wenn die falsche Münze ganz am Ende der Folge liegt - dann benötigen wir $n-1$ Vergleiche und liegen somit in $O(n)$.

b) Entwerfen und beschreiben Sie umgangssprachlich einen rekursiven Divide-And-Conquer Algorithmus, der das Problem für große n deutlich schneller löst.

Am besten geht man hier mit einer modifizierten Version der binären Suche vor. Da n immer glatt durch 2 teilbar ist, haben wir immer eine identische Menge an Münzen auf der linken und auf der rechten Seite, wenn wir die Ausgangsmenge in der exakten Mitte teilen. Nun kann man diese beiden Mengen miteinander vergleichen. Falls die Anzahl der Münzen in jeder Hälfte größer als 1 ist, muss dieser Vorgang wiederholt werden, bis nurnoch 2 Münzen miteinander verglichen werden, dann ist die leichtere der beiden die falsche Münze.

c) Analysieren Sie den Worst-Case und Best-Case Aufwand dafür. Zählen Sie die Anzahl der Wiegevorgänge dabei genau (nicht asymptotisch)

Nun unterscheidet sich der o.g. Algorithmus von der normalen binären Suche dadurch, dass wir keinen Vergleich mit einem einzelnen Element haben, mit dem wir vorzeitig abbrechen könnten. Wir müssen also jedes mal, selbst wenn die falsche Münze in der Mitte liegt, die Hälften immer ganz abarbeiten was bei 2k Elementen zu genau $log_2 n$ oder anders ausgedrückt k Vergleichen führt.

2 Modifizierter Mergesort

a) Beschreiben Sie die Arbeitsweise von $merge4$ umgangssprachlich. Bestimmen Sie die worst-case-Anzahl von Vergleichen von Vergleichen für $merge4$ auf 4 gleich großen sortierten Folgen von je m Elementen. Zählen Sie die Vergleiche genau (keine O-Notation).

Beim normalen *merge* werden zwei in sich sortierten Listen zu einer großen sortierten Liste zusammen gefügt, in dem immer das kleinere der beiden Elemente der beiden Teillisten zu erst in die neue Liste geschrieben wird, dieser Vorgang kostet einen Vergleich.

Nun müssen bei $merge4$ nicht 2 sondern 4 Listen verglichen werden, es muss also bei jedem Vorgang das minimum von 4 Elementen identifiziert werden, das kostet 3 Vergleiche. Sobald die Anzahl aller Elemente der Teillisten 4 ist brauchen wir nur 3! Vergleiche um die restlichen Elemente einzusortieren.

Also anders ausgedrückt braucht $merge4$ im Worst-Case $((m-1)*4*3)+3!$ Vergleiche. Der Worst-Case tritt ein, wenn die Elemente gleichmäßig auf die Teillisten verteilt ist, also bei $m=2$

die erste Liste $[1, 5]$, die zweite Liste $[2, 6]$, die dritte Liste $[3, 7]$ und die vierte Liste $[4, 8]$ ist. Diese Aufteilung verhindert das vorzeitige Abarbeiten einer einzelenen Liste wodurch Vergleiche eingespart werden könnten.

b) T(n) sei die worst-case Anzahl von Vergleichen des Sortierverfahrens auf einer Inputfolge der Länge n. Nehmen Sie an, dass n ein Vielfaches von 4 ist. Stellen Sie die Rekursionsgleichung für T(n) auf.

$$\begin{aligned} T(4) &= 0 \\ T(n) &= 4T(\frac{n}{4}) + merge4(\frac{n}{4}) \end{aligned}$$

$$merge4(m) = ((m-1) * 4 * 3) + 3!$$

c) Lösen Sie die Rekursionsgleichung asymptotisch mit dem Master-Theorem

$$\begin{aligned} T(n) &= aT(\frac{n}{b}) + merge4(\frac{n}{b}) \\ merge4(m) &\in \Theta(m^d) \\ &\Rightarrow a = 4 \\ &\Rightarrow b = 4 \\ &\Rightarrow c = 1 \\ &\Rightarrow d = \log_b a \\ &\Rightarrow T(n) \in \Theta(n \log_2 n) \end{aligned}$$

d) Wird diese Variante in der Praxis schneller sein als der Standard-Mergesort?

Nein. Der Aufwand der einzelenen Merge-Vorgänge ist deutlich höher, viel mehr Elemente gleichzeitig betrachtet werden müssen. Grob betrachtet ist der Aufwand für $merge2$ für 2 Elemente der Länge $m = 2m - 1$ der Aufwand für $merge4$ war ja bereits oben als $((m-1) * 4 * 3) + 3!$ definiert. Also bei 16 Elementen wäre der Aufwand von $merge2(8) = 2 * 8 - 1 = 15$ und der Aufwand von $merge4(4) = ((4-1) * 4 * 3) + 3! = 36 + 6 = 42$. Die größten Bestandteile der Laufzeit von $merge2$ bzw. $merge4$ sind bei $merge2 = 2m$ und bei $merge4 = 12(m-1)$ – somit wird $merge4$ in der Praxis eigentlich immer langsamer sein.

3 Umdrehen von Listen

```
def reverse(head):
    first = copy(head)

    while first.next is not Null:
        old_head = copy(head)
        head = first.next
        first.next = head.next
        head.next = old_head

    return head
```

4 Datenstrom-Analyse

Sie sollen die Umsätze eines Web-Shops kontinutierlich überwachen und erhalten zu jedem Verkauf den Umsatz als Zahl übermittelt. Ihr Input ist also ein beliebig langer fortlaufender Strom von Zahlen. Weiterhin wird vom Anwender beim Start des Algorithmus eine natürliche Zahl $k > 1$ gewählt. k bleibt während der Laufzeit unverändert. Das zu lösende Problem besteht aus zwei Teilproblemen:

1. Jede neu eingehende Zahl einzeln geeignet verarbeiten, damit Sie die Anfragen in 2. beantworten können.
2. Jederzeit auf Anfrage die k kleinsten alle rbisher empfangenen Umsatzzahlen in irgendeiner Reihenfolge auszugeben. (Sie können davon ausgehen, dass Anfragen erst gestellt werden, wenn schon mindestens k Zahlen eingegangen sind.)

a) Entwerfen und beschreiben Sie umgangssprachlich einen Algorithmus, der die beiden Teilprobleme unter Verwendung einer Prioritätswarteschlange effizient löst. Die Laufzeit Ihres Algorithmus darf nur von k abhängen, nicht aber von der Anzahl der empfangenen Umsatzzahlen! Brauchen Sie dafür eine Maximum- oder Minimum-Prioritätswarteschlange?

Es muss lediglich eine einzige Maximum-Prioritätswarteschlange der Länge k gepflegt werden. Sind noch weniger als k Zahlen in der Prioritätswarteschlange, kann die Zahl in jedem Fall einfach über $insert()$ eingefügt werden.

Ansonsten muss die eintreffende Zahl mit der jeweils höchsten Zahl bzw. dem Root der Prioritätswarteschlange verglichen werden. Wenn die neue Zahl kleiner als die alte Root ist, qualifiziert sich die neue dafür in den k kleinsten der bisher eingetroffenen Zahlen zu sein. Wenn die eintreffende Zahl qualifiziert ist, kann sie über $insert()$ eingefügt werden und anschließend kann die größte Zahl über $removeMax()$ entfernt werden.

Sobald eine Anfrage eintrifft, die k kleinsten Zahlen auszugeben, gibt es 2 Möglichkeiten. Entweder es wird eine Referenz zur Prioritätswarteschlange zurückgegeben, was zwar sehr effizient ist, aber unter Umständen ungewünschte Seiteneffekte nach sich ziehen könnte, falls jemand von ”außen” in die Prioritätswarteschlange schreibt. Alternativ, könnte eine ineffizientere, aber sichere Variante sein, eine Kopie der Inhalte der Prioritätswarteschlange zu erstellen und diese zurückzugegeben - so können ungewünschte Seiteneffekte vermieden werden, außerdem würden

sich die Daten des Abfragenden nach der Abfrage nicht mehr ändern. Würde die Referenz zurückgegeben werden, müsste quasi nur ein einziges Mal eine Anfrage gestellt werden, da jedes Mal über die Referenz zugegriffen werden könnte.

Im Zuge der Sicherheit würde ich zu der langsameren aber stabilieren Variante - der Kopie - raten.

b) Analysieren Sie den asymptotischen Aufwand Ihres Algorithmus für Teilproblem 1 und 2 in Abhängigkeit von k.

Der Vorgang einen Umsatz zu empfangen, liegt im BestCase in $\Omega(1)$ (falls die Zahl größer als das alte Maximum ist). Im WorstCase muss der neue Wert eingefügt und der alte Wert entfernt werden. Ein $insert()$ in eine Prioritätswarteschlange kostet $log_2 k$ und ein $removeMax()$ kostet $2log_2 k$. Somit ist der Gesamtaufwand im WorstCase ca. $3log_2 k$ und liegt daher in $O(log_2 k)$. Der AverageCase dürfte in der Praxis weit unter $log_2 k$ liegen (natürlich total davon abhängig davon wie viele Zahlen aus welchem Zahlenbereich), da bei unendlich vielen Zahlen die Wahrscheinlichkeit zunehmend abnimmt, dass eine Zahl erfasst wird, die kleiner als die höchste der bisherigen Zahlen ist.

Die Ausgabe der k kleinsten Zahlen bei einer Anfrage zu tätigen, muss die k Zahlen lange Prioritätswarteschlange kopiert werden - dies liegt in $\Theta(k)$.

Um nun eine Zusammenfassende Auskunft über den tatsächlichen AverageCase zu treffen fehlen Eckdaten, wie der (bisherige) maximale Umsatz, häufigkeit der Abfragen und Wahrscheinlichkeit bestimmter Umsatzgrößen.