Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Department Informatik
Markus Püschel
Peter Widmayer
Thomas Tschager
Tobias Pröger
Tomáš Gavenčiak

24. November 2016

Algorithmen & Datenstrukturen — Blatt P10 — HS 16

Abgabe: Bis zum 24. November 2016 um 10 Uhr auf dem Judge (ausschliesslich Quellcode).

Aufgabe P10.1 *Teilarray finden.*

Gegeben sind ein Array $A = \{a_0, \ldots, a_{n-1}\}$ und ein kürzeres Array $B = \{b_0, \ldots b_{k-1}\}$ mit Ganzzahlen. Ihre Aufgabe besteht darin, alle Positionen in A zu finden, bei denen B vorkommt. Wir sagen, dass B in A an Position i vorkommt, wenn für alle $j = 0, \ldots, k-1$ gilt, dass $b_j = a_{i+j}$.

Die offensichtliche Lösung wäre, für jedes $i = 0, \ldots, n-k$ zu prüfen, ob B in A an Position i vorkommt, indem man alle k Elemente von B vergleicht. Diese Lösung hätte aber eine Laufzeit in $\mathcal{O}(nk)$ im schlimmsten Fall. Das ist zu langsam.

Wir schlagen eine Lösung vor, die Hashing benutzt. Für eine Hash-Funktion H berechnen wir den Hash $H(B)$ von B, und anschliessend für jedes $i = 0, \ldots, n-k$ den Hash $H(a_i a_{i+1} \ldots a_{i+k-1})$. Wenn dieser Hash gleich ist wie $H(B)$, vergleichen wir $(a_i a_{i+1} \ldots a_{i+k-1})$ und B Element für Element, um sicherzugehen, dass der Hash nicht zufälligerweise gleich ist, was bei jeder Hash-Funktion passieren kann. Wir geben i aus, falls alle verglichenen Elemente gleich sind.

Wenn wir nun $H(a_i a_{i+1} \ldots a_{i+k-1})$ für jedes i von Grund auf neu berechnen, hat unsere Lösung immer noch eine Laufzeit in $\mathcal{O}(nk)$. Wir können allerdings eine "Sliding window Hash-Funktion" benutzen, die Folgendes erlaubt: Wenn wir $H(a_i a_{i+1} \ldots a_{i+k-1})$ kennen, können wir $H(a_{i+1} a_{i+2} \ldots a_{i+k})$ in Laufzeit $\mathcal{O}(1)$ berechnen. Eine solche Hash-Funktion könnte zum Beispiel die Summe der Elemente des Teilarrays sein, da die Summe aktualisiert werden kann, indem man a_i abzieht und a_{k+1} dazu zählt. Das wäre aber keine wirklich gute Hash-Funktion und wir könnten immer noch falsche Kollisionen generieren (d.h. gleiche Hash-Werte wie $H(B)$) und müssten in so einem Fall immer noch elementweise nachprüfen[1].

Eine bessere Hash-Funktion mit Parametern c und m ist die folgende:

$$H_{c,m}(a_i a_{i+1} \ldots a_{i+k-1}) = \sum_{j=0}^{k-1} a_{i+j} c^{k-j-1} \bmod m$$

Diese kann schnell für das nächste Fenster ohne a_i und mit a_{i+k} berechnet werden:

$$H_{c,m}(a_{i+1} \ldots a_{i+k}) = \sum_{j=1}^{k} a_{i+j} c^{k-j} \bmod m = c H_{c,m}(a_i \ldots a_{i+k-1}) + a_{i+k} - c^k a_i \bmod m$$

[1] Zum Beispiel liefert jede Permutation von Zahlen den gleichen Hash-Wert. Ausserdem ist der Wertebereich der Hash-Funktion für einige Daten zu klein.

Wir können den Hash-Wert also wie folgt aktualisieren:

```
H = (c * H + A[i+k] + (m - cToK) * A[i]) % m;
```

wobei `cToK` $= c^k \bmod m$.

Es gibt viele gute Möglichkeiten, Werte für c und m zu wählen. Für diese Aufgabe können Sie beispielsweise $m = 32768 = 2^{15}$ und $c = 1021$ (eine Primzahl) benutzen. Beachten Sie, dass Sie dafür sorgen müssen, dass der Wert den Wertebereich des Datentyps `int`, also $-2147483648 = -2^{31} \ldots 2147483647 = 2^{31} - 1$, nicht verlässt; etwa, indem Sie den Wert modulo m berechnen. Bei der Berechnung von `cToK` durch Multiplizieren müssen Sie in *jedem* Schritt modulo m rechnen[2]. Falls Sie sich wundern, warum wir `... + (m - cToK) * A[i]` statt `... - cToK * A[i]` rechnen: Dadurch bleiben die Zahlen immer positiv[3].

Weitere Informationen zu Modularer Arithmetik finden Sie beispielsweise unter `https://en.wikipedia.org/wiki/Modular_arithmetic`.

Eingabe Die Eingabe besteht aus mehreren Tests. Die erste Zeile enthält die Anzahl der Tests, die folgen.

Jede Test-Eingabe besteht aus drei Zeilen: Die erste Zeile enthält die Ganzzahlen $0 < n \leq 500\,000$ und $0 < k \leq n$, durch Leerzeichen getrennt. Die zweite Zeile enthält n Ganzzahlen $a_0, \ldots, a_{n-1}$, durch Leerzeichen getrennt. Die dritte Zeile enthält k Ganzzahlen $b_0, \ldots, b_{k-1}$, durch Leerzeichen getrennt.

In allen Tests liegen die Werte von A und B zwischen 0 und 1000 (inkl.).

Ausgabe Die Ausgabe soll für jeden Test eine einzige Zeile enthalten. Die Zeile für einen Test soll alle Positionen in A enthalten, bei welchen B vorkommt, und anschliessend das Wort `DONE`, alles durch Leerzeichen voneinander getrennt.

Bonus Sie erhalten einen Bonuspunkt pro 100 Punkte auf dem Judge (abgerundet). Insgesamt können Sie bis zu 200 Punkte auf dem Judge erhalten. Damit alle Tests auf dem Judge erfolgreich sind, sollte die Laufzeit Ihres Programms in $\mathcal{O}(n + kp)$ liegen (eine ausreichend effiziente Implementierung vorausgesetzt), wobei p die Anzahl der gefundenen Positionen ist. Beispielsweise können Sie $\mathcal{O}(n)$ Laufzeit verwenden um ungefähr p Kandidaten zu finden und anschliessend $\mathcal{O}(k)$ um diese zu überprüfen. Beachten Sie, dass Programme mit einer Laufzeit von $\mathcal{O}(nk)$ keine Punkte erhalten.

Senden Sie Ihr `Main.java` unter folgendem Link ein: `https://judge.inf.ethz.ch/team/websubmit.php?cid=18985&problem=DA_P10.1`. Das Passwort für die Einschreibung ist "`quicksort`".

[2] Verwenden Sie keine Datentypen für Gleitkommazahlen – diese sind nicht genau genug.

[3] In Java gilt `-1 % 10 == -1` und nicht `9`, wie wir es hier benötigen würden.

Beispiele

Eingabe

```
3
10 4
1 2 3 1 2 3 1 2 3 4
2 3 1 2
15 7
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 2
12 3
2 1 2 1 2 2 1 2 1 2 1 2
2 1 2
```

Ausgabe (der zweite Test enthält keine Positionen, an denen B in A vorkommt)

```
1 4 DONE
DONE
0 2 5 7 9 DONE
```

Hinweise Wir stellen für diese Aufgabe eine Programmvorlage als Eclipse Projektarchiv auf der Vorlesungswebseite zur Verfügung. In der Vorlage wird die Eingabe bereits eingelesen. Das Archiv enthält weitere Testdaten, damit Sie lokal testen können. Ausserdem stellen wir zusätzlich ein `Judge.java` Programm zur Verfügung, das Ihr Programm `Main.java` mit allen verfügbaren Testdaten testet – öffnen und starten sie dazu einfach `Judge.java` in derselben Weise wie Sie `Main.java` starten würden. Die Art und Weise, wie Ihr Programm `Main.java` arbeitet, wird dadurch nicht beeinflusst. `Judge.java` soll lediglich das lokale Testen erleichtern. Die zur Verfügung gestellten Testdaten sind nicht die Testdaten, welche der Judge verwendet, und im Vergleich nicht so umfangreich.

Wir freuen uns über Rückmeldung zum lokalen Judge `Judge.java`.