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

24th November 2016

Algorithmen & Datenstrukturen Exercise Sheet P10 AS 16

Hand-in: Before Thursday, 1st December 2016 10:00 via the online judge (source code only).

Exercise P10.1 *Finding a subarray.*

There is an array $A = \{a_0, \ldots, a_{n-1}\}$, and a shorter array $B = \{b_0, \ldots b_{k-1}\}$ of integers, and your task is to find all occurrences of B in A as an subarray. We say that B occurs in A at index i if for all $j = 0, \ldots, k-1$ it holds that $b_j = a_{i+j}$.

The obvious solution – for every $i = 0, \ldots, n-k$, check if B occurs in A at index i by comparing all k elements of B. But that would give us $\mathcal{O}(nk)$ time solution in the worst case, which is too slow.

We propose a solution with hashing: For some hash function H, compute a hash $H(B)$ of B, and then for every $i = 0, \ldots, n-k$, compute $H(a_i a_{i+1} \ldots a_{i+k-1})$. If this hash is equal to $H(B)$, compare $(a_i a_{i+1} \ldots a_{i+k-1})$ to B element by element (to make sure the hash was not the same accidentally, which may happen with any hash function) and on success report i.

Now if we recompute $H(a_i a_{i+1} \ldots a_{i+k-1})$ from scratch for every i, this will still take $\mathcal{O}(nk)$ time as before. But we may use use "sliding window hash functions" that allow the following: if we know $H(a_i a_{i+1} \ldots a_{i+k-1})$, we can compute $H(a_{i+1} a_{i+2} \ldots a_{i+k})$ in time $\mathcal{O}(1)$. One such "hash" function could be just the sum of the elements of the subarray – this can be updated just by subtracting a_i and adding a_{i+k}, but this is not a very good hash function and can still generate many false collisions (equalities with $H(B)$) that need to be checked in an expensive way[1].

A better hash function is the following, with some parameters c and m.

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

This can be computed quickly for the next window, without a_i and with a_{i+k}:

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

In other words, to update the hash, you can use the expression

[1] For example, any permutation of numbers generates the same hash. Also, the hash value range might be too small for some data.

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

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

Many choices of c and m are good, but here you may use for example $m = 32768 = 2^{15}$ and $c = 1021$ (a prime number). Note that you need to make sure that the value does not overflow the range of `int` type, which is $-2147483648 = -2^{31} \ldots 2147483647 = 2^{31} - 1$, but computing modulo m takes care of that. As a technical note, you need to do the modulo m in *every* step when computing `cToK` by multiplication[2]. And if you wonder why we do `... + (m - cToK) * A[i]` instead of `... - cToK * A[i]`, that is to keep the numbers always positive (with the intended result)[3].

For more reading on modular arithmetic see for example `https://en.wikipedia.org/wiki/Modular_arithmetic`.

Input The input consists of several cases. The first line of the file contains the number of cases to follow.

Each case consists of three lines: The first line contains the integers $0 < n \leq 500\,000$ and $0 < k \leq n$, separated by a space. The second line contains n integers $a_0, \ldots, a_{n-1}$, separated by spaces. The third line contains k integers $b_0, \ldots, b_{k-1}$, separated by spaces.

In all cases, the values of A and B are between 0 and 1000 (inclusive).

Output For every case, the output should contain a single line with the indexes of all the positions of B in A, and then the word `DONE`, all separated by spaces.

Grading You will get 1 bonus point for every 100 judge points, rounded down. Your may get up to 200 judge points. The program should be reasonably efficient and work in $\mathcal{O}(n + kp)$ time to get full points, where p is the number of found matches – e.g. you can use $\mathcal{O}(n)$ time to find approximately p match candidates and then time $\mathcal{O}(k)$ to verify each of them. Note that program with complexity $\mathcal{O}(nk)$ might get no points.

Submit your `Main.java` at `https://judge.inf.ethz.ch/team/websubmit.php?cid=18985&problem=DA_P10.1`, enroll password is "`quicksort`".

Examples

Input

```
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
```

Output (the second case contains no matches)

[2] Avoid using floating point types – they are not precise enough.

[3] In Java, `-1 % 10 == -1` and not `9` as we would need.

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

Notes For this exercise, we provide a program template as an Eclipse project archive on the lecture website, which will load the input for you. The archive also contains more test data for your local testing. You can use the provided `Judge.java` program to run your `Main.java` on your computer on all the provided tests – just open and run `Judge.java` in the project as you would run `Main.java`. This does not change the way that your `Main.java` works and is just an extra tool. Also, the local test data are of course different and generally much smaller than the data that are used in the online judge.

Please let us know if you have any feedback on the provided local Judge.