# Rabin Karp

## Description

This algorithm leverages the power of hash functions to speed up string searching. The idea is to be able to hash a part of a string in a special way so you can in constant time calculate a hash function for the same string without the first character, and in constant time calculate the hash function for the same string with an extra character at the end. Then to run the algorithm, you find the hashcode of the needle, and compare it to the hashcode of all the needle length substrings of the haystack. This is easy to do because we have constructed our hashfunction in such a way that we can add and remove characters from it in constant time. So it should be possible to "advance" the hash function one position in the haystack in constant time. Then, if the hash functions are ever the same, you do the brute force check to see if the needle actually occurs at that position in the haystack.

The challenge of this algorithm lies in creating the hash function. If we were able to create a perfect hash function, then the running time would be $O(n+m)$. Practically that is the running time of the algorithm, but in the worst case Rabin Karp can take $O(nm)$ time.

## Hash Function

The most commonly used hash function treats the substring as a number in a large base. Generally the base chosen is a large prime number. So for the string "apple", let's compute the hash function of substrings of length 4, using the base 1337.

$$\begin{aligned}
\texttt{hash(appl)} &= \texttt{a}\cdot 1337^3 + \texttt{p}\cdot 1337^2 + \texttt{p}\cdot 1337^1 + \texttt{l}\cdot 1337^0 \\
&= 97\cdot 1337^3 + 112\cdot 1337^2 + 112\cdot 1337^1 + 108\cdot 1337^0 \\
&= 232028393621
\end{aligned}$$

$$\begin{aligned}
\texttt{hash(pple)} &= \left(\texttt{hash(appl)} - \texttt{a}\cdot 1337^3\right)\cdot 1337 + \texttt{e}\cdot 1337^0 \\
&= \left(\texttt{a}\cdot 1337^3 + \texttt{p}\cdot 1337^2 + \texttt{p}\cdot 1337^1 + \texttt{l}\cdot 1337^0 - \texttt{a}\cdot 1337^3\right)\cdot 1337 + \texttt{e}\cdot 1337^0 \\
&= \left(\texttt{p}\cdot 1337^2 + \texttt{p}\cdot 1337^1 + \texttt{l}\cdot 1337^0\right)\cdot 1337 + \texttt{e}\cdot 1337^0 \\
&= \texttt{p}\cdot 1337^3 + \texttt{p}\cdot 1337^2 + \texttt{l}\cdot 1337^1 + \texttt{e}\cdot 1337^0 \\
&= 267878084561
\end{aligned}$$

You can see from the above equations, you can compute `hash(pple)` in constant time if you have `hash(appl)`. You just subtract the first letter, multiply the result by the base, and add the new letter.

```
int hash(char[] str, int length, int base) {
   int hash = 0;
   for (int m = 1, int i = length - 1; i >= 0; i--, m *= base) {
      hash += str[i] * m;
   }
   return hash;
}

/**
 * @param hash    the old hash function
 * @param base    the base being used
 * @param power   base raised to the power of needle.length - 1
 * @param remove  the character to remove from the front
 * @param add     the character to add to the back
 */
int update(int hash, int base, int power, char remove, char add) {
   return (hash - (remove * power)) * base + add;
}
```

## Algorithm

The algorithm is very simple once you understand the hash function, simply iterate over the haystack updating the hash function as you go. If you find a substring that has the same hash as the needle you do the brute force comparison to see if it is actually a match.

```
int indexOf(char[] needle, char[] haystack) {
    int base = largeRandomPrime();
    int power = power(base, needle.length - 1);
    int hash = hash(needle, needle.length, base);
    int prev = hash(haystack, needle.length, base);
    if (prev == hash && check(needle, haystack, 0) {
        return 0;
    }
    for (int i = 1; i < haystack.length - needle.length + 1; i++) {
        prev = update(prev, base, power, haystack[i - 1], haystack[i +
            needle.length - 1]);
        if (prev == hash && check(needle, haystack, i)) {
            return i;
        }
    }
    return -1;
}

boolean check(char[] needle, char[] haystack, int start) {
    if (start + needle.length - 1 >= haystack.length) {
        return false;
    }
    for (int i = 0; i < needle.length; i++) {
        if (needle[i] != haystack[i + start]) {
            return false;
        }
    }
    return true;
}
```