32. String Matching

There four string matching algorithms that will be discussed. Each of them differ in performance.

Algorithm	**Preprocessing Time**	**Processing Time**
Naive	0	$O((n-m+1)m)$
Rabin Karp	$\Theta(m)$	$O((n-m+1)m)$
Finite automaton	$O(m\|\sum\|)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Omega(m)$	$\Theta(n)$

Where, **n** is length of term, **m** is length of pattern, and **Σ** is number of alphabet that formed pattern.

Naive string-matching algorithm

The naive algorithm finds all possible matched for each character in pattern to each character in term. Because of this, total processing time is $O((n-m+1)m)$. The algorithm described below:

```
n = t.length
m = p.length
for i=0 to n:
  if [$t_i$, …, $t_{i+m}$] == [$p_0$,…, $p_m$]
    // Matched
```

Exercise:

32.1-1 Show the comparisons the naive string matcher makes for the pattern P = 0001 in the text T = 000010001010001.

Matched in shift: 1, 5, 11

32.1-2 Suppose that all character in the pattern P are different. Show how to accelerate naive string matcher to run in time $O(n)$ on n-character text T.

```
n = t.length
m = p.length
k = m - 1
for i=0 to n:
  if t[i] == p[j]:
    if j == k:
      // Matched
      j = 0
    j++
  else:
    if j > 0
      j = 0
```

32.1-3 Suppose that pattern P and text T are randomly choosen strings of length m and n, respectively, form *d*-array alphabet $\Sigma_d = \{0, 1, \ldots, d-1\}$, where *d >= 2*. Show that the expected number of character-to-character made by the implicit loop in line 4 of the naive algorithm is:

$$(n-m+1)\frac{1-d^{-m}}{1-d^{-1}} \leq 2(n-m+1)$$

over all execution of this loop. (Assume that the naive algorithm stops comparing characters for a given shift once it finds a mismatch or matches the entire pattern.) Thus, for randomly chosen strings, the naive algorithm is quite efficient.

32.1-4 Suppose we allow the pattern P to contain occurrences of **gap character** - that match an arbitrary string of characters (even one of zero length). For example, the pattern *ab-ba-c* occurs in the text *cabccbacbacab* as:

c ab cc ba cba c ab
  ab - ba - c

and as

c ab cccbac ba ______ c ab
  ab - ba - c

Note that the gap character may occur an arbitrary number of times in the pattern but not at all in the text. Give a polynomial-time algorithm to determine whether such a pattern P occurs in a given text T, and analyze the running time of your algorithm

Rabin-Karp algorithm

Rabin-Karp algorithm purposed hash function to identify pattern in given set of random number. We used Rabin-karp algorithm to solve string matching problem by **treated any character as number**. That's would be easy, since all character have been mapped into **ascii standart**. For example, character 'a' is mapped to number 97, and 'b' mapped to 98. To built a hash function, we should understand polynomial computation. Lets take a look this polynomial computation below:

$$p(x)=a_n x^n+a_{n-1}x^{n-1}+\dots+a_1 x+a_0$$

$$p(x)=\sum_{i=0}^{n} a_i x^i=a_0+a_1 x+a_2 x^2+a_3 x^3+\dots+a_n x^n$$

Hash of pattern "abc" in 26 character set is:

$$p(\text{abc})=97*26^2+98*26+99$$

We can say that p(abc) has highest order equal to two and hash value equal 68219. So, if we try to find "*abc*" in the random set "*cgdabcef*", we need to calculate possible hash value of each character in "*cgdabcef*" which matched to hash value of P(abc).

Lets take a look for an easy example:

Find pattern "abc" in set "cgdabcef"

Let:

d is how many atomic term

n is length of set *"cgdabcef"*

m is length of patten *"abc"*

h is highest order of pattern

Hp is hash value of pattern

Hs(k, m-1) is hash value of set from index k to index m-1

Initialization:

d = 26

n = 8

m = 3

h = 26^2 = 676

Hp = P(abc) = 68219

Hs(k, m-1) = ?

Matching:

for 0 to m:

1. Hs(0, m-1) = 69706
2. Hs(1, m-1) = 72325
3. Hs(2, m-1) = 70220
4. Hs(3, m-1) = 68219
5. Hs(4, m-1) = 68923
6. Hs(5, m-1) = 69652

Result:

Found match in shift 3.

If we take a look for section matching in the example above, there look such heavy computation because we need to calculate hash value in each shift. We could optimize calculation of rehashing by implementing polynomial evaluation using **Horner's Rule**:

$$p(x)=\sum_{i=0}^{n} a_i x^i = a_n + x(a_{n-1} + x(a_{n-2} + \ldots + x(a_2 + a_1 x)..))$$

Suppose, we need to move from $p(i)$ = [a_i , a_k] into $p(i+1)$ = [ai_{-1} , a_{k+1}]:

$$p(i+1)=x(p(i)-a_i x^{k-1})+a_{k+1}$$

For example, we already know that p(cgd) is 69706. Then to calculate next window, we need to calculate p(gda) that equal to **26 * (69706 – (97 * 26^2)) + 97 = 72325**.

Another optimization is by using modulo to avoiding very long calculation. Since we know that that:

$$a \equiv b \pmod{n}$$

We can replace **(97 * 26^2)** from example before, into **(97 * 26^2) mod 97**. I used 97 here because intuitively we can say that any prime number is good to be **common divisor**.