Lecture 11 : String Matching

Justin Pearson

December 6, 2022

Today's topics

- Naive string matching algorithm and analysis
- Rabin-Karp algorithm for faster String Matching.

Two things to take away from today's lecture.

- Even something as simple searching for a sub-string can be improved.
- The core idea of the Rabin-Karp algorithm is using a fingerprint computed with a hash function to search for things. With some imagination the technique can be used in other domains.

Sub-string search or Matching

Given some longer string:

> *We are such stuff as dreams are made on, and our little life is rounded with a sleep.*

Find all occurrences of some sub string "life".

> *We are such stuff as dreams are made on, and our little* **life** *is rounded with a sleep.*

Applications and algorithms

Even though it is a simple problem it has had a lot of attention. Most people use string search everyday in their text editor. Finding sub-strings in DNA sequences can have important consequences in biology.
There are a lot of algorithms on the market:

- Naive Algorithm : just search through the string.
- Optimal skipping algorithms (Knuth-Morris-Pratt and Boyer-Moore algorithms). When you make a mismatch try to work out how far ahead you can skip.
- Various algorithms based on finite automata.
- Hashing based (Rabin-Karp) today's main algorithm
- Tries are special tree based data structures based on prefixes in a string.

Naive Brute Force Method

Find $P[0], \ldots, P[m-1]$ in $T[0], \ldots, T[n-1]$.

for $s = 0, \ldots, n-m$ **do**
  $j \leftarrow 0$
  while $T[s+j] = P[j]$ **do**
    $j \leftarrow j+1$
    if $j = m$ **then**
      return s
    end if
  end while
end for
return -1.

Naive Brute Force Method

Find "ABRA":

s	j	0	1	2	3	4	5	6	7	8	9	10
	txt $\rightarrow$	A	B	A	C	A	D	A	B	R	A	C
0	2	A	B	R								
1	0		A									
2	1			A	B							
3	0				A							
4	1					A	B					
5	0						A					
6	4							A	B	R	A	

Red denotes a mismatch.

Complexity Analysis

Worst Case:

- Outer loop $n - m + 1$ iterations
- Inner loop max m constant-time iterations

Total $(n - m + 1)m = O(nm)$ as $m \leq n$.

Worst case behaviour

Find "AAAAB" in "AAAAAAAAAB"

s	j	0	1	2	3	4	5	6	7	8	9
	txt $\rightarrow$	A	A	A	A	A	A	A	A	A	B
0	4	A	A	A	A	B					
1	4		A	A	A	A	B				
2	4			A	A	A	A	B			
3	4				A	A	A	A	B		
4	4					A	A	A	A	B	
5	5						A	A	A	A	

In general the naive algorithm is bad with patterns containing repeats. In a lot of biological applications there will be lots of repeated patterns. Try to think about how you would optimise the naive algorithm so that you could skip ahead optimally (or backup as little as possible).

Using Fingerprints

The bottleneck in the naive algorithm is using the inner loop. You cannot really avoid going through the n positions of $P[0], \ldots, P[n-1]$ but we can optimise the inner loop.

General idea assume that we have a finger print function , f, that takes m characters then we can loop P and compare the fingerprint of $f(P[s], \ldots, P[s+m-1])$ with $f(T[0], \ldots, T[m-1])$.

We want out fingerprint function to be correct that is

$$X[0] \cdots X[m-1] = Y[0] \cdots Y[m-1]$$

if and only if

$$f(X[0], \ldots, X[m-1]) = f(Y[0], \ldots, Y[m-1])$$

Fingerprints

Find $P[0], \ldots P[m-1]$ in $T[0], \ldots, T[n-1]$.

$f_P \leftarrow f(P[0], \ldots, P[m-1])$.
for s in $0 \ldots n-m$ **do**
  if $f_P = f(T[s], \ldots, T[s+m-1])$ **then**
    return s
  end if
end for
return -1

Fingerprints

The catch is that useful fingerprint functions generally take $O(m)$ steps to evaluate.

For the fingerprint function to be useful it must depend on all the characters. If it did not then it would not work for all strings

It is both easy and hard (depending on how much you think about it) that a good fingerprint function must look all the characters in the string, and so a lower bound on its complexity will be $\Omega(m)$.

Proving this rigorously is very hard, you have to work out what your assumptions are.

Fingerprints

So we need a finger print function where we can calculate $f(T[s], \ldots, T[s+m-1])$ from $f(T[s-1], \ldots, T[s-1+m-1])$ in constant time.

$f_P \leftarrow f(P[0], \ldots, P[m-1])$ — $O(m)$
$f_T \leftarrow f(T[0], \ldots, T[m-1])$ — $O(m)$
for s in $0 \ldots n-m$ **do**
    if $f_P = f_T$ **then**
        return s
    end if
    $f_T \leftarrow f(T[s+1], \ldots, T[s+1+m-1])$ using f_T in constant time — $O(1)$
end for
return -1

Horner's Method

There is a trick find fingerprint functions where we can compute $f_T \leftarrow f(T[s+1], \ldots, T[s+1+m-1])$ using f_T in constant time.

There is a trick known as Horner's method that was invented much earlier by Lagrange (everything named after somebody was usually discovered by someone else), and it can be found in ancient Chinese and Persian mathematics texts.

It is a fast way of evaluating polynomials that has many applications.

The polynomial

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

equals

$$a_0 + x(a_1 + x(a_2 + x(a_3 + \cdots + x(a_{n_1} + x a_n) \cdots)))$$

You only need n multiplications and n additions.

Horner's method

Given a base 10 number say

$$56232 = 10000 \cdot 5 + 1000 \cdot 6 + 100 \cdot 2 + 10 \cdot 3 + 2$$

Then 62323 is

$$(56232 - 10000 \cdot 5) \cdot 10 + 3$$

You are shifting the digits by using properties of base 10 representations. Or you can think of your base 10 number as a polynomial.

Horner's method

Given two base R numbers:

- $x_i = t_i R^{M-1} + t_{i+1} R^{M-2} + \cdots + t_{i+M-1} R^0$
- $x_{i+1} = t_{i+1} R^{M-1} + t_{i+2} R^{M-2} + \cdots + t_{i+M} R^0$

Then

$$x_{i+1} = (x_i - t_i R^{M-1}) R + t_{i+M}$$

Rolling Fingerprints

Assume that $f(t_0, \ldots, t_{m-1}) = t_0 \cdot R^{M-1} + t_1 \cdot R^{M-2} + \cdots t_{m-1} R^0$, where R is the size of our alphabet.

$f_P \leftarrow f(P[0], \ldots, P[m-1])$.
$f_T \leftarrow f(T[0], \ldots, T[m-1])$.
for s in $0 \ldots n-m$ **do**
    if $f_P = f_T$ **then**
        return s
    end if
    $f_T \leftarrow (f_T - T[s]R^{M-1})R + T[s+m]$.
end for
return -1

If all goes well the running time is $O(n)$.

What's the catch?

With a large character set and long patterns then $f(T[s], \ldots, T[s+m-1])$ gets very large.

We cannot assume that $f(T[s], \ldots, T[s+m-1])$ fits into one machine word (32 bits, 64 bits or whatever). This means that we cannot assume that the arithmetic works in $O(1)$ time.

You would have to implement some multi-precision library and the computation time would depend on the size of the number that you are computing.

Modular Arithmetic

Use the ring of integers mod k.
We say that

$$a \equiv b \mod k$$

if there exists some integer c such that $a - b = ck$.

For example $38 \equiv 14 \mod 12$ because $38 - 14 = 24 = 2 \cdot 12$.

Pick a good value of k, do your arithmetic mod k and everything will fit in a machine word.

For example $6 +_{12} 10 = 16 \equiv 4 \mod 12$.

Fingerprint algorithm using Modular Arithmetic

Assume that $f(t_0, \ldots, t_{m-1}) = t_0 \cdot R^{M-1} +_k t_1 \cdot R^{M-2} +_k \cdots t_{m-1} R^0$, where R is the size of our alphabet.

$f_P \leftarrow f(P[0], \ldots, P[m-1])$.
$f_T \leftarrow f(T[0], \ldots, T[m-1])$.
for s in $0 \ldots n-m$ **do**
    if $f_P = f_T$ **then**
        return s
    end if
    $f_T \leftarrow (f_T -_k T[s]R^{M-1})R +_k T[s+m]$.
end for
return -1

What's the catch?

Fingerprint algorithm using Modular Arithmetic

When we were not using modular arithmetic then we had the property that:

$$X[0] \cdots X[m-1] = Y[0] \cdots Y[m-1]$$

if and only if

$$f(X[0], \ldots, X[m-1]) = f(Y[0], \ldots, Y[m-1])$$

If we are doing modular arithmetic mod k then on most inputs $k << R^m$. This is a simple application of the pigeon hole principle. You are trying to put a large number of items in a smaller number of boxes, this means that some boxes will have more than one item in.

Pigeon Hole Principle[1]

[1] Picture taken from Wikipedia

Recovering from the catch

With our modular arithmetic version we still have that:

$$X[0] \cdots X[m-1] = Y[0] \cdots Y[m-1]$$

implies

$$f(X[0] \cdots X[m-1]) = f(Y[0] \cdots Y[m-1])$$

So if

$$f(X[0] \cdots X[m-1]) \neq f(Y[0] \cdots Y[m-1])$$

then

$$X[0] \cdots X[m-1] \neq Y[0] \cdots Y[m-1].$$

This means that we only have to check for mistakes when the fingerprints are equal.

Hash Functions

The general theory is complicated, but hash functions appear all over computer science.

If you do arithmetic modulo some prime number q then you reduce the number of unnecessary clashes.

Then the fingerprint function

$$f(t_0, \ldots, t_{m-1}) = t_0 R^{m-1} +_q \cdots +_q t_{m-1}$$

is not such a bad hash function.

If q is the about mn^2 then the probability of a false collision is about $1/n$.

Rabin-Karp Algorithm

Finding P in T.

$q \rightarrow$ a large enough prime.
$f_P \leftarrow f(P[0], \ldots, P[m-1])$.
$f_T \leftarrow f(T[0], \ldots, T[m-1])$.
for s in $0 \ldots n-m$ **do**
    if $f_P = f_T$ **then**
        if $P[0..m-1] = T[s..s+m-1]$ **then**
            return s
        end if
    end if
    $f_T \leftarrow (f_T -_k T[s]R^{M-1})R +_k T[s+m]$.
end for
return -1

Rabin-Karp Algorithm

- Again we back to the worst-case running time of $O(nm)$ but that is only if you make false matches.
- As above If you pick a large enough prime, but not too large (to avoid overflow), then the probability of collision is going to be $1/n$.
- This means the expected running time of the algorithm is roughly $O(n)$, because the $1/n$ probability will result in the average only one collision per loop through n. This is not really true, but a good enough lie.
- Easier to implement and less memory intensive than some of the alternatives.
- Lots of extensions of the same idea: two-dimensional searching, multiple pattern search. Using a hash function to filter out things that are not equal quickly is a powerful idea.

Alternatives

- Knuth-Morris-Pratt and Boyer-Moore optimal skipping.
- Automaton based methods.
- Regular expression matching (extensions of Automaton).
- Approximate matching.