Lecture 8: Hashing in String Matching

built on 2021/01/28 at 09:47:31

There are *two* basic tasks that we're going to discuss today:

- Given a string pattern p, find all occurrences of this pattern in a text t.
- Given many string patterns $p^{(1)}, p^{(2)}, \ldots, p^{(k)}$, all of the same length, find all occurrences of these patterns in a text t.

Why do we want to do such tasks? Text editor. Grep. Virus scanner. Scanning for "taboo" words. Looking for specific things in (human) genomes.

Basic String Matching: Let's formalize this problem a little. Say the pattern is a string $p = p_1 p_2 p_3 \ldots p_m$ of length m. A text t is $t = t_1 t_2 \ldots t_n$ of length n. These are strings from some Σ^*. There is a match at position s if

$$\text{for all } i = 1, \ldots, m: \; t[s+i-1] = p[i]$$

As an example:

```
t: acbabaabcbac
p: ^^^abaa
```

1 Naïve String Matching

How do we solve this problem? The most basic solution one can think of is probably to look at every possible starting point in the text t and see if p is a match there. In code:

```
for i = 1 to n - m + 1
  // is there a match starting t[i]?
  if match(i):
     report a match at i
```

and we can implement match as follows:

```
def match(st):
   for j = 1 to m
      if t[st+j-1] != p[j]:
         return False
   return True
```

If $t =$ `acbabaabcbac` and $p =$ `abaa`, then the following comparisons will take place:

```
t: acbabaabcbac
   **
    *
     *
      ***!
       *
...
```

What's the running time of this algorithm? In the worst case, the outer loop looks at every possible starting point. There are $n - m + 1$ of them. The inner loop can potentially look at all m positions, so a total of $O(nm)$.

Is it possible to do better? You probably assume the answer is yes. To gather some thoughts, let's work through the following exercises:

- Show the comparisons the naïve algorithm makes for $p =$ `0001` and $t =$ `00001000101001`.
- Now suppose the characters in the pattern p are *all* different. How can we potentially speed up the string matching algorithm? What's the new running time?

Among other problems: we repeatedly look at some characters and while we know we can go ahead, this algorithm doesn't make use of that fact.

2 Our Old Friend: DFA

An awesome property of a DFA is that it takes time linear in the input string to go through. The real questions are, how to build one given a pattern p? And how much space is needed to keep it around?

To motivate why this is not completely trivial, consider the pattern $p =$ `ababaca`. You'll spend a moment constructing a DFA that accepts if this string is a substring (not subsequence, i.e., no jumping over characters) in the input.

Everyone knows pretty much that the main "skeleton" looks like this:

```
0 -a-> 1 -b-> 2 -a-> 3 -b-> 4 -a-> 5 -c-> 6 -a-> 7
```

The thing to ponder is, what to do say when you're in state 5 and the next character is a b.

If we number the states 0 through m, then the answer, which we can ponder later, is $\delta(i,\sigma) = \chi(p_1 p_2 \dots p_i \sigma)$, where $\chi(x)$ is the the largest k such that $p_1 p_2 \dots p_k$ is a suffix of x.

Using some high-powered technology, we can construct such a DFA in $O(m|\Sigma|)$ and spend another $\Theta(n)$ going through the input string.

Then there's Knuth-Morris-Pratt (KMP); read online to find out. KMP takes $\Theta(m)$ preprocessing time and $\Theta(n)$ to go through the input string.

3 Rabin-Karp Algorithm

We're going to revisit the naïve string matching algorithm. The main idea is to find a faster way to implement `match`, which seems impossible at first glance.

It seems impossible because at some level, checking if there is a match starting at position i—exactly what `match(i)` is doing—requires examining the whole pattern, taking $O(m)$ in the worst case.

Indeed, it is not possible to avoid this running time in the worst-case scenario. But is it possible to speed up `match` in the common case? The answer turns out to be very simple. We'll do this in two steps:

- We'll make `match` more complicated for no good reason (apparently).
- Then, we'll show how to derive benefits out of it.

Let me show you the before-and-after description for the `match` algorithm:

```
// Before:
def match(st):
   for j = 1 to m
      if t[st+j-1] != p[j]:
         return False
   return True
```

```
// After:
def match(st):
   hash_t = hash(t[st:st+m])
   hash_p = hash(p)
   if hash_t != hash_p:
      return False
   else:
      run the original match(st)
```

Indeed, this is more complicated. First, we compute the hashes for both the pattern and the slice of text being compared. If the hash values are **not** the same, we reject. But if they're the same, we resort to our good ol' `match`.

Does this give us any benefit? Not really. For now, I want to convince you that the new `match` isn't wrong:

- If the hashes are different, then there isn't a match for sure.
- If the hashes are the same, then there may or may not be a match. That's why we need to use the tried-and-true `match`.

The hope is that we can compute the hashes fast and therefore reject nonmatches quickly (which is the case that doesn't pay off). How?

Rabin Fingerprint: We'll take a bit of a detour. Let's warm up with a silly question. A hash function, for our purposes, maps a string in Σ^* to a number. Is there a hash function h such that for a character c, $h(s+c)$ can be derived from the value of $h(s)$ quickly? In more human terms, for example, if we know that $h(\text{"}ab\text{"}) = 1421$, can we compute $h(\text{"}abc\text{"})$ quickly somehow, perhaps by adjusting the value 1421? In other words, we're looking for a hash function with the property that $h(s+c) = f(h(s), c)$.

As it turns out, the hash function in question is pretty simple: Say we wish to compute the hash for a string $s = s_1 s_2 \dots s_m$. Each character s_i can be viewed as a number in some base (e.g., the ASCII code is in base 256). Then, if we pick a prime (hopefully a large one) q, then

$$h(s) = (s_1 b^{m-1} + s_2 b^{m-2} + s_3 b^{m-3} + \cdots + s_m) \,\%\, q$$

As an example, say we're using the ASCII code and the base $b = 256$. Then, if we pick $q = 983$, then the string $s = \text{"}ab\text{"}$ hashes as follows:

$$h(\text{"}ab\text{"}) = (\underbrace{97}_{\text{a}} \cdot 256^1 + \underbrace{98}_{\text{b}} \cdot 256^0) \,\%\, 983 = 24930 \,\%\, 983 = 355.$$

As another example, if we hash the string "abc", here's what we get:

$$h(\text{"}abc\text{"}) = (\underbrace{97}_{\text{a}} \cdot 256^2 + \underbrace{98}_{\text{b}} \cdot 256^1 + \underbrace{99}_{\text{c}} \cdot 256^0 +) \,\%\, 983 = 543.$$

It is not a coincidence that $h(\text{"}abc\text{"}) = (h(\text{"}ab\text{"}) \times 256 +' c') \,\%\, q$. We can show this mathematically. In the general case, we have

$$\begin{aligned} h(s+c) &= (s_1 b^m + s_2 b^{m-1} + s_3 b^{m-2} + \cdots + s_m b + c) \,\%\, q \\ &\equiv_q (s_1 b^m + s_2 b^{m-1} + s_3 b^{m-2} + \cdots + s_m b) + c \\ &\equiv_q (s_1 b^{m-1} + s_2 b^{m-2} + s_3 b^{m-3} + \cdots + s_m 1) b + c \\ &\equiv_q h(s) \cdot b + c \end{aligned}$$

This means it is easy to add one character to an existing string. How about removing the first character? This, too, isn't too hard. We'll do this by example.

Say we have $h(\text{"}abc\text{"})$ and we want to derive $h(\text{"}bc\text{"})$. Let's write out $h(\text{"}bc\text{"})$.

$$h(\text{"}bc\text{"}) = (\underbrace{98}_{\text{b}} \cdot 256^1 + \underbrace{99}_{\text{c}} \cdot 256^0 +) \,\%\, 983 = 612$$

Notice that the only difference between $h(\text{"}abc\text{"})$ and $h(\text{"}bc\text{"})$ is that the term $97 \cdot 256^2$ is dropped. This is easy to do, we know that the letter we're dropping is an "a" and the original string has length $m = 3$, so we just subtract off

$$\underbrace{97}_{\text{a}} \cdot 256^{m-1} = 97 \times 256^2.$$

Therefore, $h(\text{"}bc\text{"}) \equiv_q h(\text{"}abc\text{"}) - 97 \times 256^2 = 612$, which is the same answer as before.

So What? We'll revisit an example we looked at early on: look for $p =$ `abaa` in $t =$ `acbabaabcbac`. A few observations are in order:

- First, there is no need to compute the hash of t every time. This won't change, so just compute it once and store.
- As we "scroll" through t, what're the things we need to hash:

```
acba
 cbab  [remove a from front, add b to back]
  baba [remove b from front, add a to back]
   abaa
    baab
     ...
```

- If q is a large prime, the chance that the pattern and the text disagree while producing the same hash value will decrease. Under certain assumptions, the expected running time of Rabin-Karp is $O(n+m)$.

4 Multi-Pattern Matching

If we have multiple patterns, what can we do? Repeat any of our favorite string matching algorithm k times, one for each pattern.

OR: Say the patterns are $p^{(1)}, p^{(2)}, \ldots, p^{(k)}$. We can build a hash table (a set in Python let's say) H that stores $h(p^{(1)})$, $h(p^{(2)})$, etc. This is so that as we examine the hash on the text, we can just check if this hash is present in the H.

5 Exercise

Describe an efficient algorithm the following problem:

> Given two long strings S and T and a number $\ell > 1$, determine if they share a common substring of length ℓ.