String Matching

1 Problem Definition

- Given a (long) string T (over some (finite) alphabet Σ), and
- A (short) pattern P over the same alphabet (or subset)
- Find one or all occurrences of P in T, if any

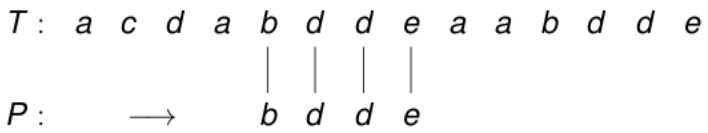
For example: T:acdabddeaabdde
P:bdde

2 Terminology

- Input is $T[1...n]$ and $P[1...m]$ with $m \leqslant n$
- Alphabet Σ can be pretty much anything
- We say a pattern P occurs with shift s in T if

$$0 \leq s \leq n - m \text{ and } T[s+1, \ldots, s+m] = P[1 \ldots m]$$

e.g.



Where we've got a shift of 4
$T[5...8] = P[1...4]$ - shift 4
$T[11...14] = P[1...4]$ - shift 10

3 Shifts

We say s is

- A valid shift is P occurs with a shift s in T, and
- An invalid shift otherwise

In the example, 4 and 10 are valid shifts, all others are not.

Goal: Find all occurrences of P in T(all valid shifts), as quickly as you can

4 More Terminology

Some notation:

- Σ is the alphabet, and Σ^* ("Sigma Star") is the transitive closure, i.e., the set of all (finite) words that you can form using characters from Σ, e.g.

$$\Sigma = \{0, 1\} \Rightarrow \Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$$

where ϵ denotes the empty word (the word of length 0)

- Length of a string x is denoted by $|x|$
- Concatenation of string x and y is denoted xy, and xy's length is $|xy| = |x| + |y|$
- Test $x \stackrel{?}{=} y$ can be done by comparing character by character starting at the leftmost position, and takes time $\Theta(t+1)$ with t being length of longest string z with $z \sqsubset x$ and $z \sqsubset y$; notice the "+1" to handle case $t = 0$ i.e. no common prefix
- Note that the notation $z \sqsubset x$ means z is the prefix of x
- Here t is the length of z (the longest match between the two strings)

5 The naive algorithm

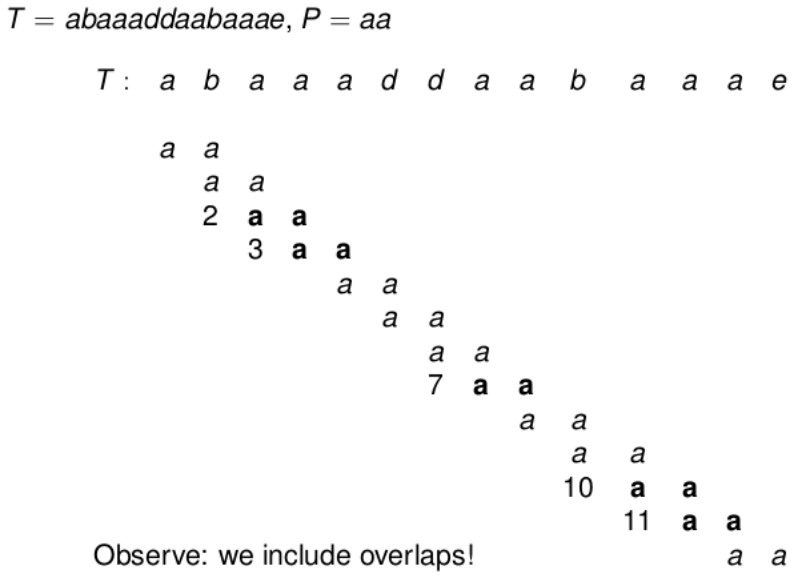
Input string $T[1...n]$, pattern $P[1...m]$
Idea simple: just slide the pattern along the string and compare at each of the $n - m + 1$ many positions

Listing 1: Naive Matcher

```
1 for s=0 to n-m do
2     if P[1...m]=T[s+1...s+m] then
3         print "pattern occurs with shift s"
4     end if
5 end for
```

Notice how we write the comparison as if it were a primitive operation(takes constant time), when in reality it isn't

6 Example



Clearly running time $O((n - m + 1) \cdot m)$
This is tight in the sense that we have worst case instances that need precisely that (consider $T = a^n$ and $P = a^m$) = we need to compare the full pattern with whatever we find at each of the $n - m + 1$ possible shifts

Notice that this is

- $\Theta(n^2)$ if $m \approx n/2$
- $O(n)$ if $m = O(1)$ or $m = n - O(1)$
- $n\sqrt{n} = n^{3/2}$ if $m \approx \sqrt{n}$

This is not particularly good

The biggest problems seems to be that we discard any information about previous comparisons when we move window to the right

6.1 Example

An example: consider $T = aaaaabaaaaa$ and $P = aaaa$

0 aaaaabaaaaa
1 aaaaabaaaaa
2 aaaaabaaaaa
3 aaaaabaaaaa
4 aaaaabaaaaa
5 aaaaabaaaaa
6 aaaaabaaaaa
7 aaaaabaaaaa

First mismatch in line 2, but do we **really** need to do 3,4, and 5?

7 The Rabin-Karp algorithm

We introduce a second string-matching algorithm:

- Still not best-possible
- Still same worst-case running time
- Still no ”intelligent skipping”
- But in practice typically much better

7.1 A useful idea

- Based on representation of substrings of length k as k-digit numbers
- That is, if our alphabet has size ten, represent string `61524` as decimal number 61,524
- This alone does of course not help - still need to make as many comparisons
- Extra idea: do calculations modulo some cleverly chosen number
- One of the very few examples where we need to think about ”real world” performance
- Would normally assume that we can do arbitrary precision arithmetic in one step
- We will assume $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- Any other alphabet size is also possible, for example binary and hex
- Rabin-Karp does some preprocessing, and then the actual matching
- All the ”better” string matching algorithms do some sort of preprocessing
- R-K needs $\Theta(m)$ time for preprocessing, and then worst case $\Theta((n - m + 1)m)$ for matching - same as naive matcher
- However, much better ”on average” - under certain assumptions

8 Decimal Representations

- Given a pattern $P[1...m]$, denote by p its decimal value
- Likewise, given string $T[1...n]$, let t_s denote decimal value of length-m substring $T = [s+1...s+m]$, for $s = 0, 1, ..., n-m$
- Clearly, $T[s+1...s+m] = P$ iff $t_s = p$
- Therefore, s is a valid shift $\Leftrightarrow t_s = p$

What if we:

- Could compute p in time $O(m)$, and
- Could compute all the t_s values in total time $O(n-m+1)$ and
- Could compare p with any given t_s in constant time, then
- We'd have a $O(m) + O(n-m+1) = O(n)$ algorithm

We can't... or can we?
We can certainly compute p from $P[1..m]$ in time $O(m)$

$$\begin{aligned}
p =& P[m] + 10P[m-1] + 100P[m-2] + 1000P[m-3] + \cdots \\
& \cdots + 10^{m-2}P[2] + 10^{m-1}P[1] \\
=& P[m] + 10(P[m-1] + 10P[m-2] + 100P[m-3] + \cdots \\
& \cdots + 10^{m-3}P[2] + 10^{m-2}P[1]) \\
=& P[m] + 10(P[m-1] + 10(P[m-2] + 10P[m-3] + \cdots \\
& \vdots \\
& = P[m] + 10(P[m-1] + 10(P[m-2] + \cdots + 10(P[2] + 10P[1]) \cdots))
\end{aligned}$$

↑ that all looks really scary, but it's just str to int conversion if you look at it
That is, use Horner's rule on $p(x) = \sum_{i=0}^{m-1} P[m-i]x^i$ in $x = 10$ (compute from the right)
Can do the same for t_0 (from T[1..m]), the first m characters of T

- We still need $t+1, t_2, .., t_{n-m}$
- We don't want to spend more than $\Theta(n-m)$ time for that, i.e. constant time for each $t_i, i \geqslant 1$

Simple, we can compute "new" t_{s+1} based on t_s:

- Consider e.g.

  15<u>6274</u>5242133
  156<u>2745</u>242133

  i.e., focus on transition for $t_s = 6274$ to $t_{s+1} = 2745$ (with m=4)
- Notice that (when looking at strings) what we're really doing is
  - Pushing the left-most 6 out, and
  - Adding the 5 as new right most character
- This corresponds to the arithmetic operation $10(6274 - 6 \cdot 1000) + 5$
- More generally, $t_{s+1} = 10\left(t_s - 10^{m-1}T[s+1]\right) + T[s+m+1]$
- Can precompute 10^{m-1} (it's always going to be 10^{m-1}), then the execution takes constant time

Summarising

- Thus, we can compute p and t_0 in time $\Theta(m)$, and can compute all $t_1, ..., t_{m-n}$ in total time $\Theta(n-m)$
- We can find all occurrences of P in T with $\Theta(m)$ for preprocessing of P, and $\Theta(n-m)$ for the matching

But wait

- Surely, if m is very big, then p and t_i are very long numbers
- We wouldn't be able to store then in a computer word each
- Even if we used arbitrarily long integers (through, say, some class or other data structure in some library), it's be unrealistic to assume that we can do the actual arithmetic in constant time

So what are we going to do

Answer is quite simple, really: do everything, i.e., all arithmetic and such, modulo some (possibly much smaller) number q, so that every number fits into one computer word

- Let's return to our example and chose $q = 123$; note $h = 10^3 \mod 123 = 16$
- Consider the string 52424537454...
- Suppose we have computed t_0 (corresponds to 5242) as 76 ($5242 \mod 123 = 76$)
- Suppose we want to compute t_1; before we just did this:

$$\begin{aligned}\mathbf{t}_1 = t_{s+1} &= 10\left(t_s - 10^{m-1}T[s+1]\right) + T[s+m+1] \\ &= 10\left(t_0 - 10^{m-1}T[1]\right) + T[0+4+1] \\ &= 10(5242 - 1000 \cdot 5) + 4] \\ &= 10 \cdot 242 + 4 = 2420 + 4 \\ &= 2424\end{aligned}$$

- Observe that $2424 \mod 123 = 87$
- Now working modulo q (recall $h = 16, t_0 = 76$)

$$\begin{aligned}\mathbf{t}_1 &= (10\,(t_0 - hT[s+1]) + T[s+m+1]) \bmod q \\ &= (10(76 - 16 \cdot 5) + 4) \bmod 123 \\ &= (10(76 - 80) + 4) \bmod 123 \\ &= (10(-4) + 4) \bmod 123 \\ &= -36 \bmod 123 = 87\end{aligned}$$

That seems to work, or does it

- Consider T=62321462338294, P=3214, and q=123
- $3214 \mod 123 = 16$
- If we compare the mod-123 values that we get for P and the substring of T at shift 2 (3214), then we get a match (which is OK)
- If we compare the mod-123 values that we get for P and the substring of T at shift 5 ($4623 \mod 123 = 72$), then we get a mismatch (which is OK)
- If we compare the mod-123 values that we get for P and the substring of T at shift 9 (3829), then we get a match (which is not OK)
- In general, $t_s \neq p \pmod q$ then we know for sure that $t_s \neq p$ (which is OK)
- However $t_s \equiv p \pmod q$ does not necessarily imply $t_s = p$ (which is not OK)

Ok, this doesn't seem to work that easily

- It looks as though we can use the text $t_s \stackrel{?}{\equiv} p \pmod q$ really only as "heuristic"
- If the answer is no then we're fine
- If the answer is yes then we'll have to check in the old fashioned way and may get a false positive
- Hope that it doesn't happen too often, so that the extra cost isn't too high
- Make q large, so that we don't get "Too many collisions"