Institut for Matematik og Datalogi
Syddansk Universitet, Odense

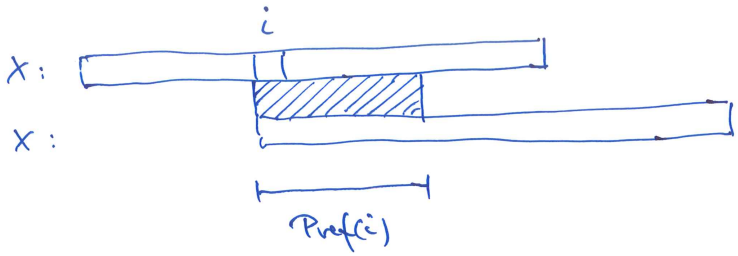
22. januar 2017
RF

The Prefix Table

As a building block in the algorithm for finding the BMShift table, the following values will prove useful. For a string X of length m (and first character indexed by zero), we define the table $\text{Pref}(i)$ of longest prefixes as follows:

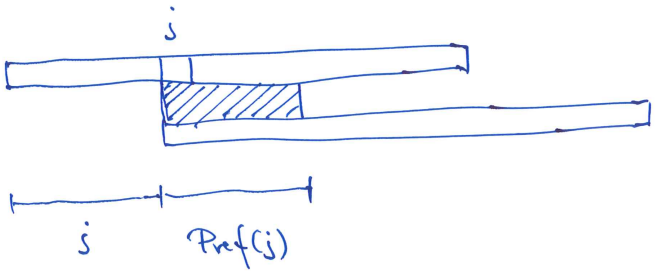
$$\text{Pref}(i) = |\operatorname{lcp}(X, X[i..(m-1)])| \text{ for } 0 \leq i \leq m$$

That is, $\text{Pref}(i)$ is the length of the longest common prefix between $X[i..(m-1)]$ and X. This is illustrated below (shaded area means matching characters):



The goal of this note is to give an algorithm for finding $\text{Pref}(i)$ for all $0 \leq i \leq m$ in time $O(m)$. The overall idea of the algorithm is to find $\text{Pref}(i)$ for increasing i, exploiting already found information when looking at the next i.

To capture the already found information, it will be useful to consider the ending point in X of the prefix in $X[j..(m-1)]$ corresponding to $\text{Pref}(j)$. This ending point is $j + \text{Pref}(j)$, as can be seen in the following figure:



Central to the algorithm is the largest such ending point found so far. We make the following definitions to describe this value and a j which generated it:

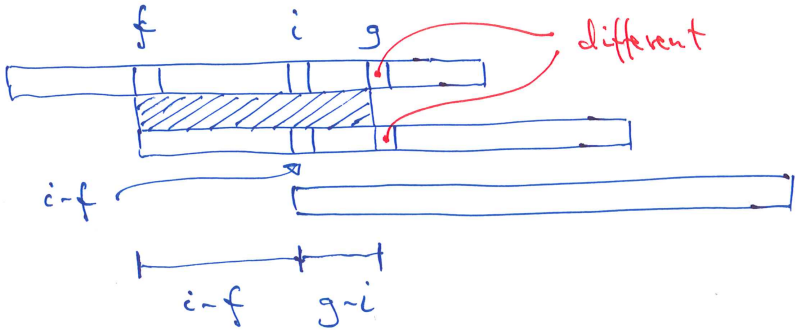
$$g(i) = \max_{0<j<i} \{j + \text{Pref}(j)\} \tag{1}$$

$$f(i) = \operatorname*{argmax}_{0<j<i} \{j + \text{Pref}(j)\} \tag{2}$$

Thus, g is the largest such ending point found so far (excluding the trivial knowledge that $\text{Pref}(0) = m$) and f is a j generating it.

For increasing i, g is clearly non-decreasing. Each iteration of the algorithm increments i, which may make g grow by some value $t_i \geq 0$. If the iteration can be performed in $O(1+t_i)$ time, the total time will be $O(\sum_{i=0}^{n}(1+t_i)) = O(n + \sum_{i=0}^{n} t_i) = O(n)$, since $\sum_{i=0}^{n} t_i$ is the final value of g, which cannot exceed n. We now describe the details of the algorithm carrying out that plan.

For an iteration generating $\text{Pref}(i)$, we assume for now that $g > i$. This situation is illustrated below (note that $f < i$ always holds by the definition of f):



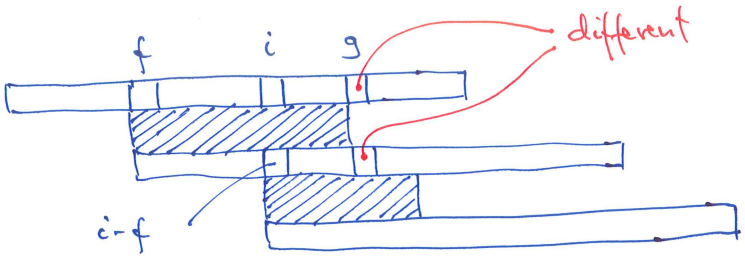
Note for later use that the two characters pointed out in the figure above must differ (else Pref(f) would be larger).

To find Pref(i), we must compare $X[i..(m-1)]$ and X, starting from the left. By the matching in the figure above we may as well compare $X[(i-f)..(m-1)]$ and X. We distinguish three cases:

Case I:	$\text{Pref}(i-f) > g-i$
Case II:	$\text{Pref}(i-f) < g-i$
Case III:	$\text{Pref}(i-f) = g-i$

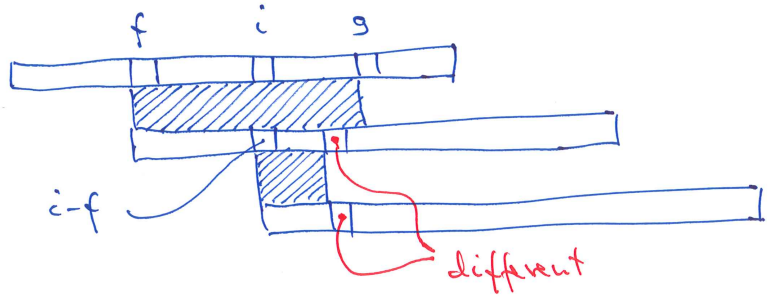
Note that by the definition of f, we have $1 \leq f \leq i-1$, hence we know $1 \leq i-f \leq i-1$. Hence, Pref($i-f$) has been found at the current time, and (assuming correct values for f and g are maintained by the algorithm) the algorithm can therefore distinguish between the cases.

Case I is illustrated below.



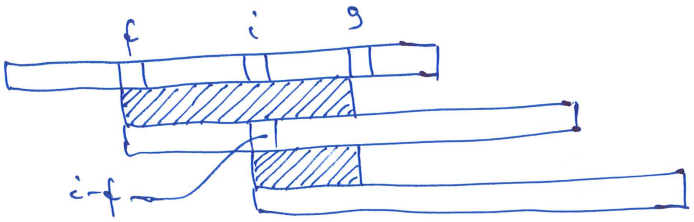
Since the two characters pointed out differ, we can deduce from the figure that Pref(i) $= g-i$, and that g and f do not change.

Case II is illustrated below.



Since the two characters pointed out differ (or Pref$(i-f)$ would have been larger), we can deduce from the figure that Pref(i) = Pref$(i-f)$, and that g and f do not change.

Case III is illustrated below.



In this case, we can only deduce that Pref$(i) \geq g - i$, and we must find how much Pref(i) is longer by comparing characters in $X[g..(m-1)]$ and $X[(i-g)..(m-1)]$ left to right. If Pref(i) ends up being increased by t_i, we have made $1 + t_i$ comparisons. The value of g has been increased by t_i, and i is the new value for f.

Here is a code implementing the above algorithmic idea:

```
Pref[0] = m
g = 0
f = 1
FOR i = 1 TO m-1
  IF i < g AND Pref[i-f] != g-i
    Pref[i] = min{Pref[i-f],g-i}
  ELSE
    g = max{g,i}
    f = i
    WHILE g < m AND X[g] == X[g-f]
      g++
    Pref[i] = g-f
```

We claim that this algorithm maintains the following invariant at the start of each iteration of the `FOR`-loop: For $0 \leq$ `j` $<$ `i`, `Pref[j]` contains the correct value Pref(`j`), and `g` and `f` obeys (1) and (2), respectively. More precisely, this invariant holds at the start of the second iteration and onwards (which is in line with (1) and (2) only being meaningful for $i \geq 2$).

We now prove the claim by induction on the `FOR`-loop. The base case is a little longer then usual: The first line clearly sets `Pref[0]` correctly to the entire string length. At the start of the first iteration, we have `i` $= 1$ and `g` $= 0$, hence the `ELSE` case is taken in this first iteration (note that the only use of the initialization `f = 1` is to make all elements of the `IF` test well-defined). After the two first lines of the `ELSE` case, we have `g` $=$ `f` $= 1$. Hence, the `WHILE`-loop is a simple left-to-right scan of $X[1..(m-1)]$ and $X[0..(m-1)]$ which correctly sets `Pref[1]` at the exit of that loop. Also at that exit, `g` and `f` obeys (1) and (2) for `i` $= 2$ (there is only one j in the maximizations in (1) and (2), namely $j = 1$). Hence, the invariant is true at the start of the second iteration of the `FOR`-loop. This proves the basis of the induction.

For the induction step, we assume the invariant is true at the beginning of one iteration, and must prove it true at the beginning of the next iteration.

If the `IF` branch is taken, we are in Case I or II above. The analysis for these cases shows that `Pref[0]` is set correctly, and that `g` and `f` does not need to change for (1) and (2) to continue to hold. Hence, the invariant is true at the start of the next iteration.

If the `ELSE` branch is taken, we separate into two cases:

Case A: `Pref`$[$`i` $-$ `f`$]$ $=$ `g` $-$ `i` and `i` $<$ `g`
Case B: `i` $\geq$ `g`

Case A is exactly Case III above. By the analysis of Case III, it is correct to set `f` to `i`. The line with `max` does not change the value of `g`. The WHILE-loop then performs the left-to-right scan from the analysis of Case III, and thus at exit leaves `g` and `Pref[i]` correctly set. Hence, the invariant is true at the start of the next iteration.

In Case B, `i` $\geq$ `g` means that we know that `i` will be the correct value for f for the new round, since `i` $+$ Pref(`i`) will be at least as large as the current value of g (this is true even if `i` $=$ `g` and it turns out that Pref(`i`) $= 0$). The code starts by raising `g` to `i`, as this is the minimal new value for `g`, and sets `f` to `i` (correct, at argued above). As both `g` and `f` are equal to `i` at the start of the `WHILE`-loop, `g-f` is zero. Thus, the `WHILE`-loop performs a left-to-right scan of $X[$`i`$..(m-1)]$ and $X[0..(m-1)]$, which is the straight-forward way to find Pref(`i`). If the `WHILE`-loop runs for t_i rounds, we have $t_i =$ Pref(`i`). Since `g` $-$ `f` $= t_i$, `Pref[i]` is correctly set. Also, `g` is left at is correct value. Hence, the invariant is seen to be true at the start of the next iteration.

This proves the induction step, and hence the invariant. From the first part of the invariant, `Pref[i]` is correct for all `i` when the algorithm terminates. Hence, the algorithm is correct.

For the time analysis, all iterations of the `WHILE`-loop will increase `g`. Nowhere in the code can `g` decrease (and it may even increase outside the `WHILE`-loop in Case B). As `g` starts out zero and cannot by larger than m (this is true for g in (1) by construction, as all values maximized over are at most m, and our invariant shows that `g` in the code fulfills (1)), at most m iterations of the `WHILE`-loop can take place in total in the algorithm. The rest of the work of the algorithm is clearly $O(m)$. In conclusion, the algorithm runs in $O(m)$ time.

We note that in the last iteration of the `FOR`-loop with `i` equal to m, there is a reference `X[m]` in the third to last code line, which is a reference to a non-existing character past the end of the string. If the programming language has short-circuiting evaluation of `AND` (such as many languages of C-ancestry, including Java) this is not a problem (the reference will not be executed). Else, the `FOR`-loop should only run until `m-1`, and `Pref[m]` should be set to its value zero separately. If the programming language has short-circuiting evaluation of `AND`, the third line of the code initializing `f` can be removed (it is only there for the first execution of the `IF` test).