11. String Search

- The goal is to find the first occurrence of a pattern P of length m in a text T of length n. Pattern P and text T can be sequences of any kind, not necessarily character sequences:

  found' = (i | 1 ≤ i ≤n–m+1 • match(i,m))
  (found' 1 ≤ i' ≤ n–m +1 match(i',m) nomatch(i'–1)

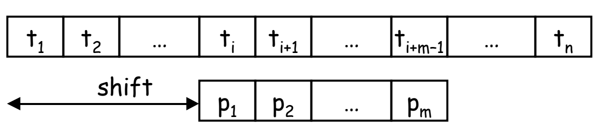
  where

  match(i,k) = (P[1..k] = T[i..i+k–1])
  nomatch(i) = (i | 1 ≤ k ≤ i •¬match(i,m))

- Chapter 34 in CLR presents three algorithms (Naive, Knuth-Morris-Pratt, Boyer-Moore) using the theory of finite state machines. Here we partly follow an alternative presentation of Wirth, Algorithms and Data Structures, Prentice-Hall, 1986, pp 56 - 69. A copy of that part of the book is in the library.

Naive String Search …

- The most straightforward solution is to start comparing P with T at position 1 and in case of mismatch shift the position of P:




```
i ← 0 ; found ← false
while ¬found    i + m ≤ n do
    ▷ invariant: nomatch(i)
    i ← i + 1
    found ← match(i, m)
```

- For the invariant, we observe that nomatch(0) holds initially and that nomatch(i–1) and ¬match(i,m) implies nomatch(i). The loop terminates with the postcondition (assuming m ≤ n):

  nomatch(i) ((¬found i+m > n) (found i+m ≤ n match(i,m))

… Naive String Search

- The statement found ← match(i,m) needs to be refined to a loop:

```
i ← 0 ; found ← false
while ¬found    i + m ≤ n do
    ▷ invariant: nomatch(i)
    i ← i + 1 ; j ← 0
    while j < m    P[j + 1] = T[i + j] do
        ▷ invariant: match(i,j)
        j ← j + 1
    found ← j = m
```


Analysis of Naive String Search

- In the average case, if the characters are drawn from an alphabet with two or more characters and occur randomly, we can expect a mismatch after less than two comparisons (cf. analysis of table search and linear search and CRL exercise 34.1-4). Hence an upper bound of the average number of comparisons is

  $2\ (n - m + 1)$

  which makes an average case running time of $O(n - m)$.

- For the worst case, suppose P consists of m−1 characters "a" followed by character "b" and
  - T consists of n characters "a", or
  - T consists of n − 1 characters "a" followed by "b".

  In both cases, $(n - m + 1)\ m$ comparisons are necessary, making a running time of $\Theta((n - m + 1)\ m)$.

Improving Naive String Search …

- The idea is to use the information provided by a partial match to avoid further comparisons which cannot possibly succeed:

Text	…	a	b	c	…		
Pattern		a	b	d			
Shifted Pattern			a	b	d		
Shifted Again				a	b	d	

Text	…	a	b	c	…		
Pattern		a	b	d			
Shifted Pattern				a	b	d	

… Improving Naive String Search

Text	…	a	a	c	…	
Pattern		a	a	d		
Shifted Pattern			a	a	d	
Shifted Again				a	a	d

Text	…	a	b	c	a	b	d	…		
Pattern		a	b	c	a	b	c			
Shifted Pattern					a	b	c	a	b	c

In other words, we could shift faster and make fewer comparisons if we know the repetitive structure of the pattern!

Structure of Knuth-Morris-Pratt Search …

				i	
T	…	a	b	c	…
P		a	b	d	
			j	j+1	

- At each position i in the text T, we compare T[i] with one or more elements of P;
- The index i used for comparisons with T[i] is either incremented by one or remains the same; it is never decremented.
- The index j used for comparisons with P[j+1] is either incremented by one or decremented by a value such that it becomes greater than or equal to zero.

183

… Structure of Knuth-Morris-Pratt Search

- The outer loop is responsible for incrementing i by one and, in case of a match, incrementing j by one. The inner loop is responsible for shifting P to the right, if possible:

```
i ← 0 ; j ← 0
while j < m    i < n do
   ▷ invariant: nomatch(i–j)    match(i–j+1, j)
   i ← i+1
   while j > 0    P[j+1] ≠ T[i] do
      j ← D
   if P[j+1] = T[i] then
      j ← j+1
found ← (j = m)
```

D[1..m]: int

j <-- D[j]

- D is still unspecified. However, we note that if D < j, then the assignment j ← D will shift P to the right! If D = 0, then the pattern is shifted beyond its current position.

184

Determining Maximal Shifts

- The idea of D is that it depends only on the pattern P and the position j, where $1 \leq j \leq m$. Hence it can be represented by D = d[j], where d is an array of type:

  d : array [1..m] of integer

- For example, for P = "ababc" we have

  d[1] = 0, d[2] = 0, d[3] = 1, d[4] = 2, d[5] = 0

  for P="ababa"?

- In general, d[j] is the length of the longest prefix of P[1..j] which is also a suffix of P[1..j]:

  $d[j] = \max\{k \mid 0 \leq k < j \quad P[1..k] = P[j-k+1..j]\}$

- Computing d amounts to searching strings, for which we can use Knuth-Morris-Pratt search itself.

185

```
......abcdefgx.......
   abcdefgy...    j = 7
      abcd...   d[j] = 3
```

Knuth-Morris-Pratt Search

```
▷ compute d
d[1] ← 0
k ← 0
for j ← 2 to m
   while k > 0   P[k+1] ≠ P[j] do
      k ← d[k]
   if P[k+1] = P[j] then
      k ← k+1
   d[j] ← k
▷ search for P
i ← 0 ; j ← 0
while j < m   i < n do
   i ← i+1
   while j > 0   P[j+1] ≠ T[i] do
      j ← d[j]
   if P[j+1] = T[i] then
      j ← j+1
found ← (j = m)
```

```
abaaaabaab...   d[9] = 4   d[4] = 1
     abaaa...   d[10] = 4+1 = 5?
        ab...   d[10] = d[4]+1
                      = 2?
```

//d[j-1] = k//

How would you analyze this algorithm? How many comparisons would it require in the worst case?

186

Principle of Boyer-Moore Search

- Knuth-Morris-Pratt search yields a genuine benefit only in the case of a partial mismatch, which is comparatively rare. Boyer-Moore Search improves also the average case.
- The idea is to start comparing the pattern with the text at the end of the pattern. In case of a mismatch, the pattern can immediately be shifted to the right by a precomputed number of positions. Example where the compared characters are underlined:

```
Hoola-Hoola girls like Hooligans
Hooligan
     Hooligan
       Hooligan
               Hooligan
                       Hooligan
```


Structure of Boyer-Moore Search

- Let match(i,j) mean that when P[1] is shifted over T[i], then all elements to the right of P[j] match the corresponding ones in T; let nomatch(i) mean that there is no complete match up to T[i]:

  match(i, j) = (P[j + 1 .. m] = T[i + j .. i + m − 1])
  nomatch(i) = (k | 1 ≤ k ≤ i • ¬match(i, 0))

- $i \leftarrow m$
  while $i \le n$ do
      ▷ invariant: nomatch(i − m)
      $j \leftarrow m$; $k \leftarrow i$
      while $j > 0$ P[j] = T[k] do
          ▷ invariant: match(i − m + 1, j) i - m = k − j
          $j \leftarrow j - 1$; $k \leftarrow k - 1$
      if j = 0 then
          return k + 1
      $i \leftarrow i + d[T[i]]$

Maximal Shifts

- d[x] is defined to be the rightmost occurrence of character x in P from the end (not including the last character):

  $(\quad k \mid m - d[x] < k < m \bullet P[k] \neq x)$

- For example, if P = "abc", then

  d[a] = 2, d[b] = 1, d[c] = 3, d[x] = 3 for all x ≠ a, b, c

- If P = "aab", then

  d[a] = 1, d[b] = 3, d[x] = 3 for all x ≠ a, b

- If P = "aba", then

  d[a] = 2, d[b] = 1, d[x] = 3 for all x ≠ a, b

Boyer-Moore Search

- Boyer-Moore-Search (P, T)

```
for each character x do
   d[x] ← m
for j ← 1 to m – 1 do
   d[P[j]] ← m – j
i ← m
while i ≤ n do
   j ← m ; k ← i
   while j > 0    P[j] = T[k] do
       j ← j – 1 ; k ← k – 1
   if j = 0 then
       return k + 1
   i ← i + d[T[i]]
```

What is the best and worst case running time?

Comparison of String Search Algorithms

- Let m be the length of the pattern and n the length of the text. We assume that the size of the alphabet is a constant (otherwise we would need to add the size to the running time of Boyer-Moore). We are interested in the average and worst case running times in case when the pattern does not occur in the text :

	Naive	Knuth-Morris-Pratt	Boyer-Moore
average	$\Theta(n)$	$\Theta(n + m)$	$\Theta(n / m)$
worst	$\Theta(n\ m)$	$\Theta(n + m)$	$\Theta(n * m)$

- Combination of Knuth-Morris-Pratt and Boyer-Moore is possible by building tables d1 and d2, respectively, and taking the larger shift of both. This way we achieve $\Theta(n / m)$ in average and $\Theta(n + m)$ in the worst case. However, the additional bookkeeping makes the gain questionable in practice.