COMP2230/6230 Algorithms

Lecture 12

Prof Ljiljana Brankovic

Lecture Overview

Text Seraching:

- R. Johnsonbaugh and M. Schaefer. *Algorithms*, Chapter 9

Lecture based on:

- R. Johnsonbaugh and M. Schaefer. *Algorithms*.
- Some slides are based on slides by Helen Giggins

Text Searching

- Common problem of retrieving information from text.

- Applications
  - Search engines
  - File management – locate a file/directory
  - Biologists searching DNA sequence data

- General problem

  Within a text ***t***, find a match for a pattern ***p***

- ***Text***, ***pattern*** and ***match*** definition can vary depending on the application.
  - ***Text*** is a document in memory (e.g., word processor).
  - ***Pattern*** is a word you want to find in the document.

Simple Text Searching

- Naïve technique to solve it – brute force sequential searching – "Simple Text Search".
- Scenario: Word processor searching for a word in document.
- Run through the text comparing the word letter by letter as we go.
- Test all locations until we find a match, or else run through the whole text and find no match.

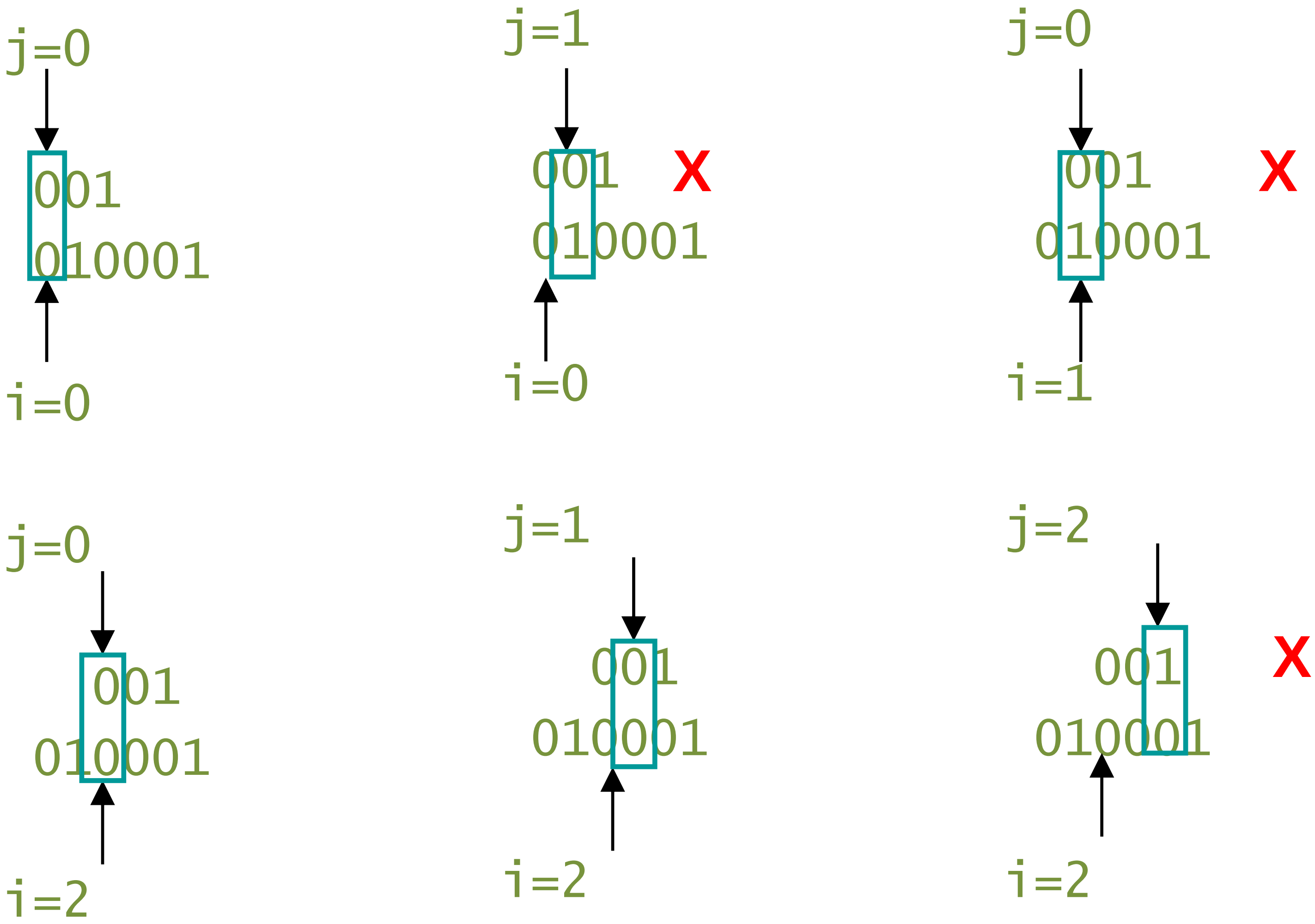
Algorithm 9.1.1 Simple Text Search

This algorithm searches for an occurrence of a pattern *p* in a text *t*. It returns the smallest index `i` such that `t [i..i+m-1] = p`, or `-1` if no such index exists.

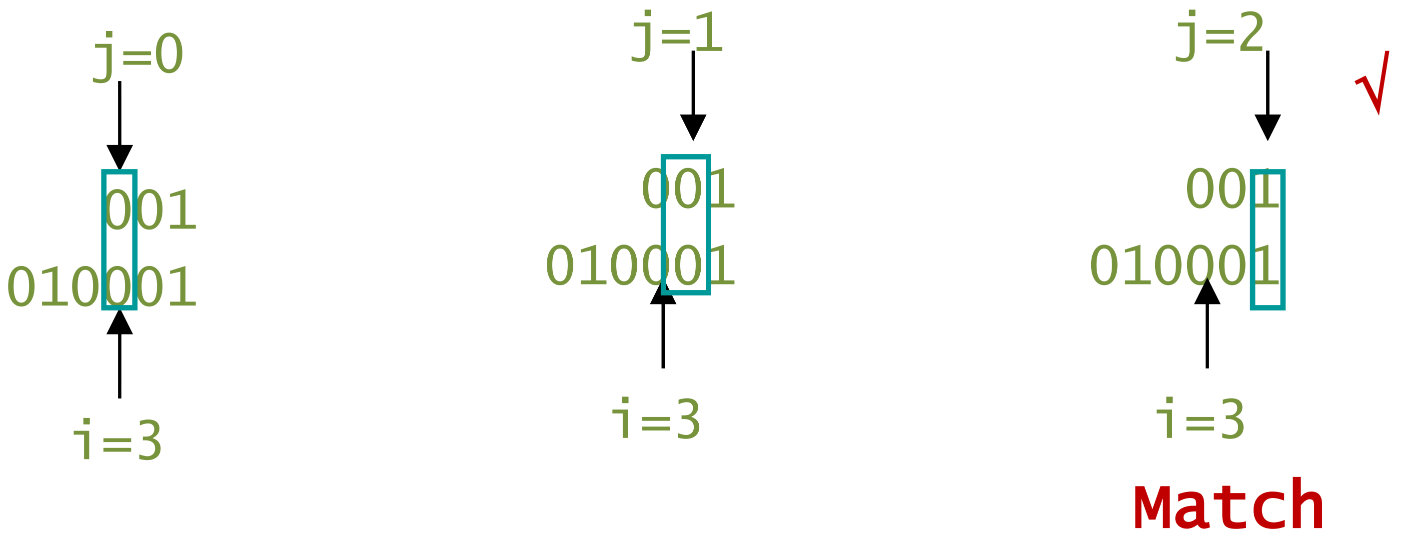
```
Input Parameters: p, t
Output Parameters: None

simple_text_search(p, t) {
    m = p.length
    n = t.length
    i = 0
    while (i + m ≤ n) {
        j = 0
        while (t [i + j] = = p [j]) {
            j = j + 1
            if (j = m)
                return i
        }
        i = i + 1
    }
    return -1
}
```

Simple Text Search

Searching for '001' in '010001'



Simple Text Search



Simple Text Search

- Algorithm is simple to understand, but not very efficient.
- Best case – the pattern is found at the beginning of the text - $\theta(m)$.
- Worst case – no pattern match – can take time $\theta(m(n-m+1))$.
- Performs better on random text and patterns $O(n-m)$.
- Simple Text Search can be ineffective for longer patterns, and particularly when text contains repeated elements. For example, when $m=\frac{n}{2}$, for the worst case we have $\theta\left(n^2\right)$.

Rabin-Karp Algorithm

- To reduce the number of comparisons for long patterns do some preliminary checks.
- What if we're searching for '0011' in '0001001000'
- Pattern has parity 0 – even number of 1s
- How many substrings can we rule out?
- There are altogether $n - m + 1 \ = \ 10 - 4 + 1 \ = \ 7$ substrings
- We only need to test one substring, at position $i \ = 3$.
- This technique is called *fingerprinting*. We only compare a small aspect of the pattern, instead of the whole pattern.

Rabin-Karp Algorithm

Fingerprinting for pattern ‘0011’ in text ‘0001001000’

i	0	1	2	3	4	5	6	7	8	9
t[i]	0	0	0	1	0	0	1	0	0	0
f[i]	1	1	1	0	1	1	1			

$f[i]$ is the parity of the string $t[i, \ldots, i+3]$, since our pattern is of length 4.

Rabin-Karp Algorithm

- We can improve this technique by applying a hash function.
- Consider the m bits to be a binary representation of a non-negative integer and take the modulo q.

$$pfinger = (\sum_{j=0}^{m-1} p[j] \times 2^{m-1-j}) mod\ q$$

- We choose q to be a prime number larger than m.

Rabin-Karp Algorithm Example 1

$text =_3$ '0001001000'

$f[i] = (\sum_{j=0}^{3} t[i+j] \times 2^{3-j}) \; mod \; 5$

$1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 9$

i	0	1	2	3	4	5	6	7	8	9
$t[i]$	0	0	0	1	0	0	1	0	0	0
$\sum_{j=0}^{3} t[i+j] \times 2^{3-j}$	1	2	4	$\boxed{9}$	2	4	8			
$f[i]$	1	2	4	4	2	4	3			

We now use the same calculation for our fingerprint.

'0011' gives $pfinger = \left(0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0\right) mod \; 5 = 3$

We only need to compare against '1000', which has $f[i]$ of 3.

Rabin-Karp Algorithm

- Worst case running time the same as for Simple Text Search: $\theta(m(n-m+1))$.
- What is the worst case?
- However, the expected running time is now $\theta(n+m)$.

Algorithm 9.2.5 Rabin-Karp Search

This algorithm searches for an occurrence of a pattern p in a text t. It returns the smallest index i such that $t[i..i+m-1] = p$, or -1 if no such index exists.

```
Input Parameters: p, t
Output Parameters: None
rabin_karp_search(p, t) {
    m = p.length
    n = t.length
    q = prime number larger than m
    r = 2^(m-1) mod q
    // computation of initial remainders
    f[0] = 0
    pfinger = 0
    for j = 0 to m-1 {
        f[0] = (2 * f[0] + t[j]) mod q
        pfinger = (2 * pfinger + p[j]) mod q}
    i = 0
    while (i + m ≤ n) {
        if (f[i] == pfinger)
            if (t[i..i+m-1] = = p) // this comparison takes O(m)
                return i
        f[i + 1] = (2 * (f[i]- r * t[i]) + t[i + m]) mod q
        i = i + 1
    }
    return -1}
```

Knuth-Morris-Pratt Algorithm

The reason why simple pattern match algorithm performs so poorly is that it tries to match the pattern even if there is no hope of matching.

Example 2.

T	H	A	N	K	S										
T	R	A	C	I	N	G	S	T	R	E	E	T	M	A	P	S
T	H	I	N	K	I	N	G	S	T	R	A	I	G	H	T

Shift table tells us by how many positions we can shift the pattern if $p[0..k]$ matches the text and $p[k+1]$ does not:

		T	H	A	N	K	S
k	-1	0	1	2	3	4	5
shift	1	1	2	3	4	5	6

Knuth-Morris-Pratt Algorithm

Example 3.

P	A	P	P	A	R																		
P	A	P	P	A	R	P	A	P	P	A	R	R	A	S	S	A	N	U	A	R	A	G	H

Shift table:

		P	A	P	P	A	R
k	-1	0	1	2	3	4	5
shift	1	1	2	2	3	3	6

Knuth-Morris-Pratt Algorithm

Input Parameters: p,t
Output Parameters: None

```
knuth_morris_pratt_search(p,t) {
    m = p.length
    n = t.length
    knuth_morrist_prath_shift(p,shift) //compute shift array
    i = 0
    j = 0
    while (i + m ≤ n){
        while ( t[i+j] = = p[j]) {
            j = j + 1
            if (j ≥ m)
                return i
        }
        i = i + shift[j-1]
        j = max(j-shift[j-1],0)
    }
    return -1
}
```

Knuth-Morris-Pratt Algorithm

Theorem.

The *knuth_morris_pratt_search* algorithm correctly computes the first occurrence of pattern p in text t in time $O(m+n)$.

Knuth-Morris-Pratt Algorithm

Input Parameters: p
Output Parameters: *shift*

```
knuth_morris_pratt_shift(p,shift) {
    m = p.length
    shift[-1]=1 // if p[0] ≠ t[i] we shift by one position
    shift[0]=1
    i = 1
    j = 0
    while (i + j < m)
        if (p[i+j] = = p[j]) {
            shift[i+j]=i
            j = j + 1
        }
        else {
            if (j = = 0)
                shift[i]=i+1
            i = i + shift[j-1]
            j = max(j-shift[j-1],0)

        }
}
```

Knuth-Morris-Pratt Algorithm

Example 4.

Trace knuth_morris_pratt_shift algorithm on pattern "papper".

				shift[-1]=1
				shift[0]=1
i=1, j=0:	p[1]≠p[0],	j=0	→	shift[1]=2
i=2, j=0:	p[2]=p[0]		→	shift[2]=2
i=2, j=1:	p[3]≠p[1],	j≠0		
i=3, j=0:	p[3]=p[0]		→	shift[3]=3
i=3, j=1:	p[4]=p[1]		→	shift[4]=3
i=3, j=2:	p[5]≠p[2],	j≠0		
i=5, j=0:	p[5]≠p[0]	j=0	→	shift[5]=6

Knuth-Morris-Pratt Algorithm

Theorem.

The knuth_morris_pratt_shift algorithm correctly computes the shift array in time $O(m)$.