CSE 5311 Notes 14: Sequences

(Last updated 4/19/17 3:56 PM)

Pattern-Based Preprocessing

Simple Rescanning

Pattern - m symbols   Text - n symbols

```
                      0 1 2 3
Example:   Pattern:   A B A C
           Text:      A B A B A B A C A B A B A C A B A
                      0 1 2 3
                        0
                          0 1 2 3
                            0
                              0 1 2 3
                                0
                                  0 1
                                    0
                                      0 1 2 3
                                        0
                                          0 1 2 3
                                            0
                                              0 1
```

Worst-Case: $\Theta(mn)$

```
                      0 1 2 3
           Pattern:   0 0 0 1
           Text:      0 0 0 0 0 0 0 0 0 0 1
                      0 1 2 3
                        0 1 2 3
                          0 1 2 3
                            0 1 2 3
                              0 1 2 3
                                0 1 2 3
                                  0 1 2 3
                                    0 1 2 3
```

Rabin-Karp Algorithm

Concept:

1. Compute signature of all m symbols of pattern. (Taking $\theta(m)$ time.)

2. Compute signature of each m contiguous symbols of text. (Taking <u>overall</u> $\theta(n)$ time.)

If function values for 1. and 2. are ever equal, then do symbol-by-symbol test.

http://ranger.uta.edu/~weems/NOTES5311/karp-rabin.c

```
#include <stdio.h>
#include <string.h>

#define B 131
#define EOS 0
#define TRUE 1
#define FALSE 0

void search(pat,text)
char *pat,*text;
{

int hpat,htext,Bm,j,m;

if (pat[0]==EOS) return;
Bm=1;
hpat=htext=0;

for (m=0;text[m]!=EOS && pat[m]!=EOS;m++)
{
  Bm*=B;
  hpat=hpat*B+pat[m];
  htext=htext*B+text[m];
  printf("%c ",text[m]);
  if (pat[m+1]==EOS)
    printf("%d",htext);
  else
    printf("\n");
}

if (text[m]==EOS && pat[m]!=EOS) return;

for (j=m;TRUE;j++)
{
  if (hpat==htext && strncmp(text+j-m,pat,m)==0)
    printf("<<<\n");
  else
    printf("\n");
  if (text[j]==EOS) return;
  htext=htext*B-text[j-m]*Bm+text[j];
  printf("%c %d ",text[j],htext);
}
}

main()
{
char pat[80],text[80],*pos,*work;

printf("Enter pattern & text\n");
while (scanf("%s %s",pat,text)!=EOF)
{
  search(pat,text);
  printf("Enter pattern & text\n");
}
}
```

Knuth-Morris-Pratt Scan Technique

Takes O($m + n$) time

Uses two tables for matcher

Pattern
Fail links

Matcher properties

Never backs up in the text
May use fail links to back up in pattern

	Pattern	Fail 1	Fail 2
0	A	−1	−1
1	A	0	−1
2	A	1	−1
3	A	2	−1
4	B	3	3

Text:	A	A	C	A	A	A	A	A	B	A	A	A
States (1):	0	1	2	0	1	2	3	4	4			
			1					3				
			0									
			−1									

Text:	A	A	C	A	A	A	A	A	B	A	A	A
States (2):	0	1	2	0	1	2	3	4	4			
			−1					3				

	Pattern	Fail 1	Fail 2
0	A	−1	−1
1	A	0	−1
2	B	1	1
3	A	0	−1
4	A	1	−1
5	B	2	1
6	A	3	−1
7	A	4	−1
8	A	5	5
9	B	2	1

Text:	A	A	B	A	A	B	A	A	B	A	A	A	B	A	A	B	A	A	A	B
States (1):	0	1	2	3	4	5	6	7	8	6	7	8	9	3	4	5	6	7	8	9
									5				2							

Text:	A	A	B	A	A	B	A	A	B	A	A	A	B	A	A	B	A	A	A	B
States (2):	0	1	2	3	4	5	6	7	8	6	7	8	9	0	1	2	3	4	5	2
									5				1						1	
													−1							

Knuth-Morris-Pratt Failure Link Construction

Fail link - seeks to reuse largest possible *suffix* before present position that matches a *prefix* of pattern.

Style 1: Choose maximum value of k with $0 \le k < j$ such that

for $0 < i \le k$: pattern[k - i] == pattern[j - i]

Now set fail[j] = k

Style 2: Choose maximum value of k with $0 \le k < j$ such that

pattern[k] != pattern[j], and

for $0 < i \le k$: pattern[k - i] == pattern[j - i]

Now set fail[j] = k

Direct application of these definitions could take $\Theta(m^3)$ time!

Either style fail link table may be constructed in $\Theta(m)$ time.

For style 1:

Suppose fail links 0 through j have already been set and fail[j] == k.

If, in addition, pattern[j] == pattern[k] then

Set fail[j+1] = k+1

But, what if pattern[j] != pattern[k]?

Move k back to fail[k] and recheck for pattern match

```
 0 b -1        11 b  6
 1 a  0        12 a  4
 2 b  0        13 b  5
 3 b  1        14 a  6
 4 a  1        15 b  7
 5 b  2        16 b  8
 6 a  3        17 a  9
 7 b  2        18 b 10
 8 b  3        19 a 11
 9 a  4        20 b  ?
10 b  5
```

For style 2:

Suppose fail links 0 through j have already been set and k is the *required maximum value* that was used when setting fail[j].

If, in addition, pattern[j] == pattern[k] then

If pattern[j+1] != pattern[k+1]

Set fail[j+1] = k+1

Else

Set fail[j+1] = fail[k+1]

But, what if pattern[j] != pattern[k]?

Move k back to fail[k] and recheck for pattern match

```
 0 b -1        8 b  1       16 b  1
 1 a  0        9 a  0       17 a  0
 2 b -1       10 b -1       18 b -1
 3 b  1       11 b  6       19 a 11
 4 a  0       12 a  0       20 b  ?
 5 b -1       13 b -1
 6 a  3       14 a  3
 7 b -1       15 b -1
```

	Fail 1	Pattern	Fail 2
0		A	
1		A	
2		B	
3		A	
4		A	
5		B	
6		A	
7		A	
8		A	
9		B	

	Fail 1	Pattern	Fail 2
0		A	
1		B	
2		A	
3		A	
4		B	
5		A	
6		B	
7		A	
8		A	
9		B	
10		A	
11		A	
12		B	

http://ranger.uta.edu/~weems/NOTES5311/KMP.c

```
/* Determine all possible occurences of pattern string in text
string using KMP technique*/
#include <stdio.h>
#include <string.h>

#define MAXPATLEN 80
#define EOS 0

void preprocpat1(pat,next)
/* Produces slower failure links, but capable of continuing after
   match */
char *pat;
int next[];
{
int i,j;
i=0;
j=next[0]= -1;
do
{
  if (j==(-1)||pat[i]==pat[j])
  {
    i++;
    j++;
    next[i]=j;
  }
  else
    j=next[j];
} while (pat[i]!=EOS);
printf("Fail link table 1\n");
for (i=0;pat[i]!=EOS;i++)
  printf("%d %c %d\n",i,pat[i],next[i]);
}

void preprocpat2(pat,next)
/* Produces faster failure links, but INcapable of continuing after
   match */
char *pat;
int next[];
{
int i,j;
i=0;
j=next[0]= -1;
do
{
  if (j==(-1)||pat[i]==pat[j])
  {
    i++;
    j++;
    next[i]=(pat[j]==pat[i]) ? next[j] : j;
  }
  else
    j=next[j];
} while (pat[i]!=EOS);
printf("Fail link table 2\n");
for (i=0;pat[i]!=EOS;i++)
  printf("%d %c %d\n",i,pat[i],next[i]);
}

void match(text1,pat)
char *text1,*pat;
{
int next1[MAXPATLEN],next2[MAXPATLEN],i,j;
char *text;
char printSymbol=' ';

if (*pat==EOS) return;
preprocpat1(pat,next1);
preprocpat2(pat,next2);

printf("%s\n",text1);
text=text1;
```

```
for (j=0; *text!=EOS;)
  if (j==(-1) || pat[j]== *text)
  {
    if (pat[j+1]==EOS)
      printSymbol='^';
    if (pat[j+1]==EOS && *(text+1)!=EOS)
      j=next1[j];  /*restart*/
    else
    {
      printf("%c",printSymbol);
      printSymbol=' ';
      text++;
      j++;
    }
  }
  else
    j=next2[j];
printf("\n");
}

main()
{
char pat[80],text[80];

printf("Enter text & pattern\n");
/*variable text is string 1, variable pat is string 2*/
while (scanf("%s %s",text,pat)!=EOF)
{
  match(text,pat);
  printf("Enter text & pattern\n");
}
}
```

KMP Complexity

m pattern symbols, n text symbols

Matcher - every comparison is preceded by a movement of one or both pointers

Text pointer always moves forward $\Rightarrow \Theta(n)$

Pattern pointer

Moves forward once for each text symbol $\Rightarrow \Theta(n)$

Total number of backward movements ≤ total number of forward movements
$\Rightarrow O(n)$ (could set up potential function based on pattern position)

Failure table construction

Lead pointer always moves forward $\Rightarrow \Theta(m)$

Prefix pointer

Moves forward once for each pattern symbol $\Rightarrow \Theta(m)$

Total number of backward movements ≤ total number of forward movements
$\Rightarrow \Theta(m)$ (could set up potential function based on pattern position)

Overall: $\Theta(m+n)$

Aside: Worst-case number of comparisons (fail 2 links) when processing a text symbol is bounded by $1 + 1.44 \lg m$.

Fibonacci strings may be used as difficult cases:

F_1 = "a" F_2 = "b" $F_n = F_{n-1}F_{n-2}$

F_3 = "ba"

F_4 = "bab"

F_5 = "babba"

F_6 = "babbabab"

F_7 = "babbababbabba"

Other Applications

Is W constructed by repetition? i.e. $W = X^k$ where X is the *smallest* such string.

String length must be factorable, then use KMP (either method) and use pointers at end.

	Pattern	Fail 2
0	0	-1
1	0	-1
2	1	1
3	0	-1
4	0	-1
5	2	2
6	0	-1
7	0	-1
8	1	1
9	0	-1
10	0	-1
11	2	2

12/(12-6)=2

	Pattern	Fail 2
0	1	-1
1	2	0
2	3	0
3	4	0
4	1	-1
5	2	0
6	3	0
7	4	0
8	1	-1
9	2	0

10/(10-6)=2.5

Are two strings (of same length) “circularly equal”?

1. Text = string2 || string 2
2. Pattern = string1
3. Does pattern occur in text?

Aside: CLRS 32.4 Prefix Function

Use Pascal, not C, array conventions:

	Pattern	Fail 1	Fail 2	π
1	a	0	0	0
2	b	1	1	0
3	a	1	0	1
4	b	2	1	2
5	a	3	0	3
6	b	4	1	4
7	a	5	0	5
8	b	6	1	6
9	c	7	7	0
10	a	1	0	1
11	?	2		

$\pi[i] = \text{Fail1}[i+1] - 1$

S. Faro & T. Lecroq, “The Exact Online String Matching Problem: A Review of the Most Recent Results”, *ACM Computing Surveys* 45 (2), Feb 2013:
`http://dl.acm.org.ezproxy.uta.edu/citation.cfm?doid=2431211.2431212`

SUFFIX ARRAYS: TEXT-BASED PREPROCESSING

Concept: Preprocess text and handle different patterns on-the-fly.

Simpler and more compact than other structures, such as suffix trees.

Sort array (`sa`) of pointers/subscripts based on corresponding suffixes of a text.

```
     0  1  2  3  4  5  6  7  8  9 10 11
s    a  b  c  d  a  b  c  d  a  b  c \0

sa  11  8  4  0  9  5  1 10  6  2  7  3
```

(These notes use null-terminated strings. Other terminators, e.g. `$`, would change the results . . .)

Key-comparison sorts can construct in $O\left(n^2 \log n\right)$ time (expected time for `qsort`):

```
int suffixCompare(const void *xVoidPt,const void *yVoidPt) {
// Used in qsort call to generate suffix array.
int *xPt=(int*) xVoidPt,*yPt=(int*) yVoidPt;

return strcmp(&s[*xPt],&s[*yPt]);
}

scanf("%s",s);
n=strlen(s)+1;
for (i=0;i<n;i++)
  sa[i]=i;
qsort(sa,n,sizeof(int),suffixCompare);
```

The *rank* of a suffix is its position in the suffix array (i.e. these are inverses: `rank[sa[i]]==i`).

The *longest common prefix* (`lcp[i]`) is the number of prefix matches for `sa[i-1]` and `sa[i]`.

i	sa	suffix	lcp	s	rank	lcp[rank]
0	11		-1	a	3	7
1	8	abc	0	b	6	6
2	4	abcdabc	3	c	9	5
3	0	abcdabcdabc	7	d	11	4
4	9	bc	0	a	2	3
5	5	bcdabc	2	b	5	2
6	1	bcdabcdabc	6	c	8	1
7	10	c	0	d	10	0
8	6	cdabc	1	a	1	0
9	2	cdabcdabc	5	b	4	0
10	7	dabc	0	c	7	0
11	3	dabcdabc	4		0	-1

U. Manber and E. Myers, "Suffix Arrays: A New Method for On-Line String Searches", *SIAM J. on Computing* 22, 5 (1993), 935-948. `http://epubs.siam.org.ezproxy.uta.edu/doi/pdf/10.1137/0222058`

J. Kärkkäinen et.al., "Linear Work Suffix Array Construction", *J. ACM* 53, 6 (Nov. 2006), 918-936 achieves linear time. `http://dl.acm.org.ezproxy.uta.edu/citation.cfm?doid=1217856.1217858`

S.J. Puglisi et.al., "A Taxonomy of Suffix Array Construction Algorithms", *ACM Computing Surveys* 39 (2), June 2007. `http://dl.acm.org.ezproxy.uta.edu/citation.cfm?doid=1242471.1242472`

> *"Impressive as the progress has been, ingenious as the methods have been, there still remains the challenge to devise a SACA that is lightweight, linear in the worst case, and fast in practice."*

T. Kasai et.al., "Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications", p. 181-192 in A. Amir and G.M. Landau (Eds): *CPM 2001*: *12th Ann'l Symp. on Combinatorial Pattern Matching*, Lecture Notes in Computer Science 2089, Springer-Verlag, 2001.

M.I. Abouelholda, et.al., "Replacing Suffix Trees with Enhanced Suffix Arrays", *J. of Discrete Algorithms* 2 (2004), 53-86. `http://www.sciencedirect.com.ezproxy.uta.edu/science/journal/15708667/2/1`

G.Navarro, "Spaces, Trees, and Colors: The Algorithmic Landscape of Document Retrieval on Sequences", *ACM Computing Surveys* 46 (4), March 2014. `http://dl.acm.org.ezproxy.uta.edu/citation.cfm?doid=2597757.2535933`

Manber/Myers method uses $O(\log n)$ radix sorts each taking $O(n)$ time to achieve $O(n \log n)$ time.

```
radixsort i=1
POS                   MSD  LSD - start of pass
  0 abcdabcdabc        98   99 ⇐ a is 97 in ASCII
  1 bcdabcdabc         99  100 ⇐ b is 98 in ASCII
  2 cdabcdabc         100  101
  3 dabcdabc          101   98
  4 abcdabc            98   99
  5 bcdabc             99  100
  6 cdabc             100  101
  7 dabc              101   98
  8 abc                98   99
  9 bc                 99  100
 10 c                 100    1
 11                     1    0 ⇐ 0 is used to pad
```

```
POS                   MSD  LSD - after radix sort
 11                     1    0
  0 abcdabcdabc        98   99
  4 abcdabc            98   99
  8 abc                98   99
  1 bcdabcdabc         99  100
  5 bcdabc             99  100
  9 bc                 99  100
 10 c                 100    1
  2 cdabcdabc         100  101
  6 cdabc             100  101
  3 dabcdabc          101   98
  7 dabc              101   98
6 buckets
```

```
POS                   MSD  LSD - after renumbering
 11                     0
  0 abcdabcdabc         1
  4 abcdabc             1
  8 abc                 1
  1 bcdabcdabc          2
  5 bcdabc              2
  9 bc                  2
 10 c                   3
  2 cdabcdabc           4
  6 cdabc               4
  3 dabcdabc            5
  7 dabc                5
```

```
radixsort i=2
POS                   MSD  LSD - start of pass
  0 abcdabcdabc         1    4
  1 bcdabcdabc          2    5
  2 cdabcdabc           4    1
  3 dabcdabc            5    2
  4 abcdabc             1    4
  5 bcdabc              2    5
  6 cdabc               4    1
  7 dabc                5    2
  8 abc                 1    3
  9 bc                  2    0
 10 c                   3    0
 11                     0    0
```

```
POS                   MSD  LSD - after radix sort
 11                     0    0
  8 abc                 1    3
  0 abcdabcdabc         1    4
  4 abcdabc             1    4
  9 bc                  2    0
  1 bcdabcdabc          2    5
  5 bcdabc              2    5
 10 c                   3    0
  2 cdabcdabc           4    1
  6 cdabc               4    1
  3 dabcdabc            5    2
  7 dabc                5    2
8 buckets
```

```
POS                   MSD  LSD - after renumbering
 11                     0
  8 abc                 1
  0 abcdabcdabc         2
  4 abcdabc             2
  9 bc                  3
  1 bcdabcdabc          4
  5 bcdabc              4
 10 c                   5
  2 cdabcdabc           6
  6 cdabc               6
  3 dabcdabc            7
  7 dabc                7
```

```
radixsort i=4
POS                   MSD  LSD - start of pass
  0 abcdabcdabc         2    2
  1 bcdabcdabc          4    4
  2 cdabcdabc           6    6
  3 dabcdabc            7    7
  4 abcdabc             2    1
  5 bcdabc              4    3
  6 cdabc               6    5
  7 dabc                7    0
  8 abc                 1    0
  9 bc                  3    0
 10 c                   5    0
 11                     0    0
```

```
POS                   MSD  LSD - after radix sort
 11                     0    0
  8 abc                 1    0
  4 abcdabc             2    1
  0 abcdabcdabc         2    2
  9 bc                  3    0
  5 bcdabc              4    3
  1 bcdabcdabc          4    4
 10 c                   5    0
  6 cdabc               6    5
  2 cdabcdabc           6    6
  7 dabc                7    0
  3 dabcdabc            7    7
12 buckets
```

```
POS                   MSD  LSD - after renumbering
 11                     0
  8 abc                 1
  4 abcdabc             2
  0 abcdabcdabc         3
  9 bc                  4
  5 bcdabc              5
  1 bcdabcdabc          6
 10 c                   7
  6 cdabc               8
  2 cdabcdabc           9
  7 dabc               10
  3 dabcdabc           11
```

The `lcp` array may be constructed in $O(n)$ time. (http://ranger.uta.edu/~weems/NOTES5311/LCPdemo.c)

```
void computeLCP() {
//Kasai et al linear-time construction
int h,i,j,k;

h=0;  // Index to support result that lcp[rank[i]]>=lcp[rank[i-1]]-1
for (i=0;i<n;i++) {
  k=rank[i];
  if (k==0)
    lcp[k]=(-1);
  else {
    j=sa[k-1];
    // Attempt to extend lcp
    while (i+h<n && j+h<n && s[i+h]==s[j+h])
      h++;
    lcp[k]=h;
  }
  if (h>0)
    h--;  // Decrease according to result
}
}
```

Accelerating binary searches on `sa` using `lcp`:

```
int slowSearchFirst(char* key) {
// Finds  first string >= key assisted by binary search on suffix array,
// but without using lcp.
int low,high,mid;
int i,j;

low=0;
high=n-1;
while (low<=high) {
  mid=(low+high)/2;
  j=sa[mid]; // Position in s
  i=0;       // Position in key
  // Like strcmp
  while (s[j]==key[i] && key[i]) {
    j++;
    i++;
  }
  if (key[i]==0 || s[j]>key[i])
    high=mid-1;
  else
    low=mid+1;
}

return low;
}
```

```
int fastSearchFirst(char* key) {
// Finds first string >= key assisted by binary search on suffix array.
// Uses lcp to limit char comparisons.
int low,high,mid;
int i,j;
int keyLength,lowMatches,highMatches,midMatches;

keyLength=strlen(key);
low=0;
lowMatches=0;
high=n-1;
highMatches=0;

while (low<=high) {
  mid=(low+high)/2;
  // How many strcmp matches can be salvaged?
  midMatches=(lowMatches<highMatches) ? lowMatches : highMatches;

  if (midMatches==keyLength) {
    high=mid-1;
    highMatches=(lcp[mid]<midMatches) ? lcp[mid] : midMatches;
  }

  j=sa[mid]+midMatches; // Position in s
  i=midMatches;         // Position in key
  while (s[j]==key[i] && key[i]) {
    midMatches++;
    j++;
    i++;
  }
  if (key[i]==0 || s[j]>key[i]) {
    high=mid-1;
    highMatches=(lcp[mid]<midMatches) ? lcp[mid] : midMatches;
  }
  else {
    low=mid+1;
    lowMatches=(lcp[low]<midMatches) ? lcp[low] : midMatches;
  }
}

return low;
}
```

`slowSearchFirst` uses $\Theta(|key| \bullet \log|text|)$ time, `fastSearchFirst` uses $\Theta(|key| + \log|text|)$ time.

Longest Common Substring Using Suffix Array & LCP

1. Construct text as <string1>$<string2>, then build SA and LCP arrays.
2. Scan LCP array for maximum entry for two adjacent SA entries from different strings.

```
i   sa  suffix                                      lcp  s  rank  lcp[rank]
0   30                                              -1   0  13    8
1   11  $010010100101001001                         0    1  24    7
2   10  0$010010100101001001                        0    0  5     6
3   27  001                                         1    0  14    6
4   24  001001                                      3    1  25    5
5   2   001001010$010010100101001001                6    0  6     4
6   5   001010$010010100101001001                   4    0  17    3
7   19  00101001001                                 6    1  28    2
8   14  0010100101001001                            9    0  10    2
9   28  01                                          1    1  21    1
10  8   010$010010100101001001                      2    0  2     0
11  25  01001                                       3    $  1     0
12  22  01001001                                    5    0  16    11
13  0   01001001010$010010100101001001              8    1  27    10
14  3   01001010$010010100101001001                 6    0  8     9
15  17  0100101001001                               8    0  19    8
16  12  010010100101001001                          11   1  30    7
17  6   01010$010010100101001001                    3    0  15    8
18  20  0101001001                                  5    1  26    7
19  15  010100101001001                             8    0  7     6
20  29  1                                           0    0  18    5
21  9   10$010010100101001001                       1    1  29    4
22  26  1001                                        2    0  12    5
23  23  1001001                                     4    1  23    4
24  1   1001001010$010010100101001001               7    0  4     3
25  4   1001010$010010100101001001                  5    0  11    3
26  18  100101001001                                7    1  22    2
27  13  10010100101001001                           10   0  3     1
28  7   1010$010010100101001001                     2    0  9     1
29  21  101001001                                   4    1  20    0
30  16  10100101001001                              7       0     -1
Length of longest common substring is 8
01001001
```

M.A. Babenko and T.A. Starikovskaya, "Computing Longest Common Substrings Via Suffix Arrays", *Proc. 3rd Int'l Computer Science Symp. in Russia*, CSR 2008, LNCS 5010, Springer, 64-75.

LONGEST COMMON SUBSEQUENCES

Dynamic Programming - review

Has important applications in genetics research.

```
0   1   a   2   3   b   4   c   5   d   e   6

  6   a   5   b   4   c   3   2   d   e   1
```

1. Describe problem input.

   Two sequences:

   $$X = x_1 x_2 \ldots x_m$$
   $$Y = y_1 y_2 \ldots y_n$$

2. Determine cost function and base case.

   $C(i, j)$ = length of LCS for $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$

   $C(i, j) = 0$ if $i = 0$ or $j = 0$

3. Determine general case.

   Suppose $C(i,j)$ has

   $$x_1 x_2 \ldots x_{i-1} A \qquad y_1 y_2 \ldots y_{j-1} A$$

   Since $x_i = y_j$, $C(i,j) = C(i-1, j-1) + 1$

   Now suppose $x_i \neq y_j$:

   $$x_1 x_2 \ldots x_{i-1} A \qquad y_1 y_2 \ldots y_{j-1} B$$

   But 'B' may appear in $x_1 x_2 \ldots x_{i-1}$ or 'A' may appear in $y_1 y_2 \ldots y_{j-1}$:

   $$C(i,j) = \max\{C(i, j-1), C(i-1, j)\} \text{ if } x_i \neq y_j$$

4. Appropriate ordering of subproblems.

   Before computing $C(i,j)$, must have $C(i-1, j-1)$, $C(i, j-1)$, and $C(i-1, j)$ available.

   Use $(m + 1) \times (n + 1)$ matrix to store C values.

5. Backtrace for solution – either explictly save indication of which of the three cases was used or recheck C values.

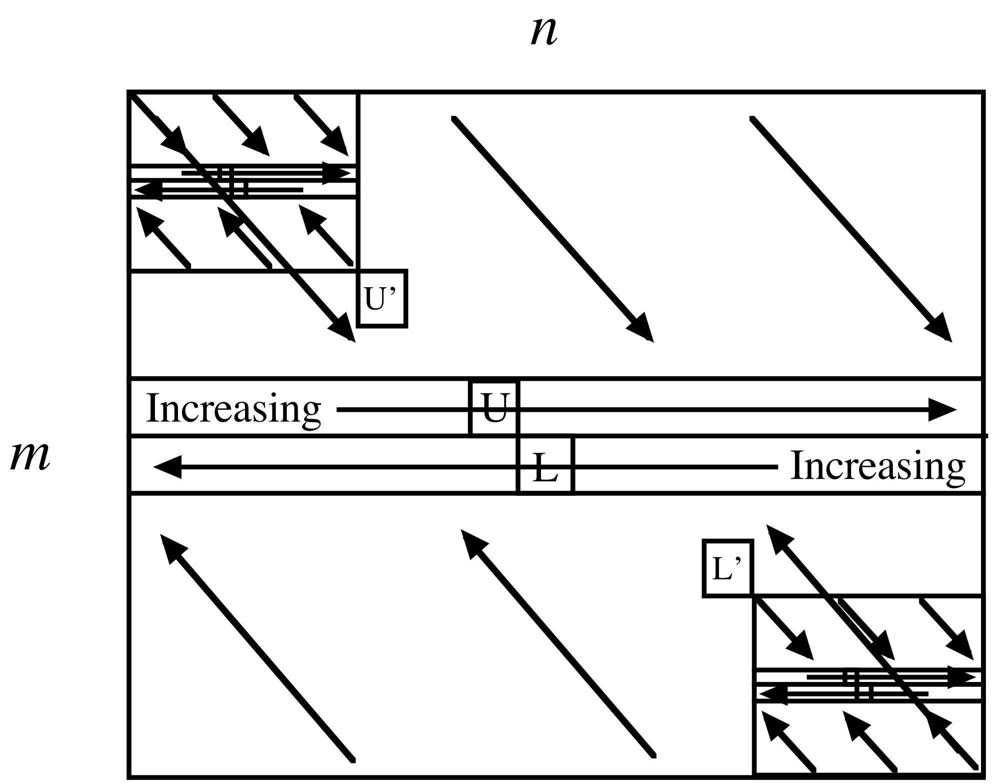
Takes $\Theta(mn)$ time.

Example:

```
010101
001100
LCS is 0100, length==4
         0  0  1  1  0  0
      0  0  0  0  0  0  0
0     0  1  1  1  1  1  1
1     0  1  1  2  2  2  2
0     0  1  2  2  2  3  3
1     0  1  2  3  3  3  3
0     0  1  2  3  3  4  4
1     0  1  2  3  4  4  4
```

Compact Version of Dynamic Programming (`http://ranger.uta.edu/~weems/NOTES5311/LCSspace.c`)

$\Omega(mn)$ space is *not* required. $O(m+n)$ space *is attainable* using a recursive algorithm.



1. Use the usual NW to SE (↘) cost computation to get increasing left-to-right cost row for LCS of the first $m/2$ elements of the first sequence and the entire second sequence.

2. Perform symmetric (SE to NW, ↖) cost computation for LCS of the last $m/2$ elements of the first sequence and the entire second sequence.

3. Scan across the two seams for the diagonal pair (U,L) with the maximum sum. This sum is the length of the LCS.

4. Scan left-to-right across the upper seam for the leftmost entry U' with the U value. If non-zero, this gives an element for the LCS output array.

5. Scan right-to-left across the lower seam for the rightmost entry L' with the L value. If non-zero, this gives an element for the LCS output array.

6. The U' and L' entries correspond to elements in the second sequence. Two simple scans can be used to find the corresponding elements of the first sequence.

7. Call recursively for each of the two subproblems. (Small subproblems may use $O(mn)$ space version.)

Still runs in $\Theta(mn)$ time:

$$T(m,n) = T\left(\frac{m}{2},a\right) + T\left(\frac{m}{2},n-a\right) + mn$$

Assume $T(k',k'') \le ck'k''$ for $k' < m$ and $k'' < n$

$$T(m,n) \le c\left(\frac{ma}{2} + \frac{m(n-a)}{2}\right) + mn = c\,\frac{mn}{2} + mn = cmn - c\,\frac{mn}{2} + mn \le cmn \text{ if } c \ge 2$$

		A	B	C	D	E	F	A	B	C	D	E	F	
	0	0	0	0	0	0	0	0	0	0	0	0	0	
A	0	1	1	1	1	1	1	1	1	1	1	1	1	
A	0	1	1	1	1	1	1	2	2	2	2	2	2	
B	0	1	2	2	2	2	2	2	3	3	3	3	3	
B	0	1	2	2	2	2	2	2	3	3	3	3	3	
C	0	1	2	3	3	3	3	3	3	4	4	4	4	
C	0	1	2	3	3	3	3	3	3	4	4	4	4	
D		4	4	4	4	3	3	3	3	3	3	2	1	0
D		4	4	4	4	3	3	3	3	3	3	2	1	0
E		3	3	3	3	3	2	2	2	2	2	2	1	0
E		3	3	3	3	3	2	2	2	2	2	2	1	0
F		2	2	2	2	2	2	1	1	1	1	1	1	0
F		1	1	1	1	1	1	1	1	1	1	1	1	0
		0	0	0	0	0	0	0	0	0	0	0	0	0

Four Russians' Method and LCS (aside, details in Gusfield)

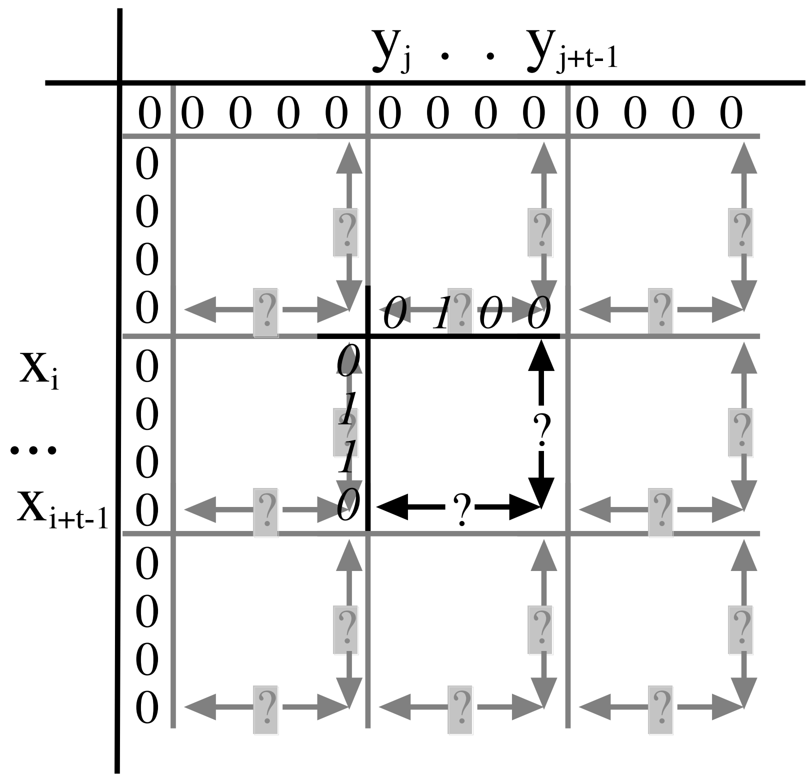
When the LCS involves a small alphabet, the following properties allow Kronrod's technique to be applied:

1. Going across a row of the $(n \times n)$ LCS matrix, $C(i,j-1) \le C(i,j) \le C(i,j-1)+1$.

2. Going down a column of the LCS matrix, $C(i-1,j) \le C(i,j) \le C(i-1,j)+1$.

(3. Going down a diagonal of the LCS matrix, $C(i-1,j-1) \le C(i,j) \le C(i-1,j-1)+1$.)

A $t \times t$ template is used to a) preprocess for all possible situations and b) tile the LCS matrix:



0 = value same as predecessor *1* = value is one more than predecessor

x_i = position in first sequence y_j = position in second sequence

? = offset values supplied by preprocessing

Typically, $t = \Theta(\log n)$ and runs in $\Theta\left(\frac{n^2}{\log^2 n}\right)$.

Sparse LCS Using Longest Strictly Increasing Subsequence (Notes 0)

0. Sparseness is the result of having a relatively large alphabet.

1. For each alphabet symbol, determine the positions (*descending* order) where the symbol appears in the second sequence.

2. Produce an intermediate sequence by *replacing* each symbol in the first sequence by its positions from the second sequence.

3. Compute a LSIS of the intermediate sequence.

4. The sequence of values from the LSIS may be used as indexes to the second sequence to obtain an LCS.

Takes time in O(r log r) where r is the length of the constructed LSIS instance.

Binary search table may be replaced by other data structures.

```
First Sequence
0 1 2 3 4 5 6 7 8

7 8 9 9 8 7 7 8 9

Second Sequence
0 1 2 3 4 5 6 7 8

7 7 8 8 6 9 8 7 9

Positions for Symbols in Second Sequence

6:  4
7:  7 1 0
8:  6 3 2
9:  8 5

Replacing Symbols in First Seq. by Positions of Second Seq.

0     1     2     3     4     5     6     7     8
7     8     9     9     8     7     7     8     9
7 1 0 6 3 2 8 5   8 5   6 3 2 7 1 0 7 1 0 6 3 2 8 5

Longest Strictly Increasing Subsequence
0 2 5 6 7 8

7 8 9 8 7 9 Longest Common Subsequence
```