Longest Common Subsequence (LCS).
A comparative study of 4 algorithms to calculate LCS.
By Soham Sadhu

Naïve recursive algorithm:

This algorithms is supposed to be the most unsophisticated of all and took the most time even for small string lengths. The algorithm rapidly degrades even if there is a small tilt towards the worst case. The program to compute the LCS length and data set generated randomly are given below.

```
/*
This program is only for the implementation of the naive recursive algorithm for the LCS problem.
With the below implementation you can only find the length of the longest subsequence but not the subsequence.
*/

// The header files included are for std io , string and file stream.
#include<iostream>
#include<string>
#include<fstream>
#include<time.h>
using namespace std;

/* The class LCS that has the two functions. One will read the strings from the file and write the LCS length.
And the other will calculate the just the LCS via naive recursive algorithm */

class LCS
{
    public:
        int lcs_len( string, string ) ;
        void lcs_read_write();
        timespec diff ( timespec , timespec ) ;
};

/* Below is the function that calculates the LCS via naive recursive algorithm */
int LCS :: lcs_len( string A, string B )
{
    if( A.length() == 0 || B.length() == 0 )
    {
        return 0;
    }
    string c, d ;
    c = A.substr( 0, A.length() - 1 );
    d = B.substr( 0, B.length() - 1 );
    if( A[ A.length() - 1 ]  == B[ B.length() - 1 ] )
    {
        return  lcs_len( c , d ) + 1 ;
    }
    else
```

```
    {
        return max( lcs_len( c, B ) , lcs_len( A, d ) );
    }
}

/* Timing function adapted from www.guyrutenberg.com/2007/09/22profiling-code-using-clock_gettime */
timespec LCS :: diff( timespec start, timespec end )
{
        timespec temp ;
        if ( ( end.tv_nsec - start.tv_nsec ) < 0 )
        {
                temp.tv_sec = end.tv_sec - start.tv_sec - 1 ;
                temp.tv_nsec = 1000000000 + end.tv_nsec - start.tv_nsec ;
        }
        else
        {
                temp.tv_sec = end.tv_sec - start.tv_sec ;
                temp.tv_nsec = end.tv_nsec - start.tv_nsec ;
        }
        return temp ;
}

/* This is the function that will read the strings from the file and write the output to console */
void LCS :: lcs_read_write()
{
    string line, A, B ;
    int position, length, lcs_length, case_num = 1 ;
    struct timespec ts, te ;
    ifstream myfile ;
    myfile.open("lcs_test_case.txt", ios::in);
    if ( myfile.is_open() )
    {
        while( myfile.good() )
        {
            getline( myfile, line );
            position = line.find(' ');
            length = line.length();
            A = line.substr( 0 , position ) ;
            B = line.substr( position + 1 , length ) ;
            clock_gettime( CLOCK_PROCESS_CPUTIME_ID, &ts ) ;
            cout << " case " << case_num << " lcs length is: " << lcs_len( A, B ) << " for strings of length " << A.length() ;
            clock_gettime( CLOCK_PROCESS_CPUTIME_ID, &te ) ;
            cout<< " and the time taken is " << diff( ts, te ).tv_sec << ":" << diff( ts, te ).tv_nsec << endl ;
            case_num++ ;
        }
        myfile.close();
    }
    else
```

```
    {
        cout << " \n Unable to open the file. The input case file may not exist. " ;
    }
}

/* The main program that will only call the read write function for LCS */
int main()
{
    LCS l ;
    l.lcs_read_write();
    return 0;
}
```

Data input set:

```
TCCGA GTCTG
TGTGCC AGTGAC
ATTGCAC CGAGCCC
GGTAGAGT CAAATAGG
TCCCACCCG CAACCTTCT
ATAAGTCGTT TTTTTAGAAC
ATGTTACACAG TTTCTAGGCTT
AGGTGGTCAGGG GGAAAAGGACGA
AACATCGGATAAG TTGTAAGATGAGC
CCATAAACAGCATG TATGACACGGCTCT
TCCAAAGCCTACTGT ATGGCCGAAGGGCCA
AAGGTATTACCCACCC AAACGCTACCACTGCG
TGCACAAGTCGACAGAC CCGCGGAGATTTGCGCC
CCCCACCGCCGCGTGCTG GGTGTACTTTGTTCGATA
GCACTGATCCCGTTAGCCG GCTAGCTTAGGCGCTCTAG
TGCAATTAATTATTCTAGTC TACTTTGCTGCTATGAAGGA
ACTCTCGGCAACGTCGCCGAA GCATTAGTTAGGTTTGCTCCT
01111 11010
010010 100101
0011111 1110010
11100000 10110101
010110011 111010010
0110110000 1101110110
01011101101 00010110110
111111000101 100101011000
0001101001101 1000100111001
10110100110101 10011011110111
111110100101101 100011111000000
1011011010001001 1001100000101110
01110110110101011 10100000110000100
101011001001010111 111100000000001000
1101010011001000010 1011000111000011001
00101101010001110011 011010000110111011100
1101101110100111110010 0010110100110001100111
```

Results:

String length	LCS length for DNA sequence	Time taken for DNA sequence	LCS length for binary sequence	Time taken for binary sequence
5	3	0.130822000	3	0.200110000
6	4	0.498650000	5	0.990300000
7	4	0.145859000	4	0.113783000
8	4	0.481506000	5	0.123907000
9	5	0.173251700	6	0.643500000
10	4	0.109669990	7	0.143757000
11	6	0.158327580	9	0.645700000
12	6	0.173810930	8	0.127210400
13	7	0.427249030	10	0.268635000
14	8	0.711489760	11	0.380082000
15	9	2.191417180	10	0.315933210
16	10	0.605278595	12	0.564631000
17	9	1.281768037	10	0.168661640
18	7	32.978360681	11	0.561077480
19	12	6.347037420	15	0.110550680
20	10	51.378347093	15	0.284645910
21	10	477.481915473	16	0.230534500

The execution time above is in seconds.

As clearly evident from the above table the recursive works well for large strings if you look at the binary string time column however it's worst case performance is exponential. For example when a string of length 21 has LCS length of 16 then it is able to complete the computation in less than a second but when the LCS length is 10 for string of length 21 then it is worst case and performance degrades rapidly to get the execution time to around 477.481915473 seconds.

The better performance for best case could be explained that once it gets the LCS character it does not make a recursion tree for that combination leaving far few combinations to compute. Any ways naïve recursive is not a good method for calculating the LCS as is evident from above.

Recursive with memoization :

Couldn't get the memoization thing to work for me; although did implement the code as was in the slides.

```
/*
this program is only for the implementation of the naive recursive algorithm with
memoization for the lcs problem.
with the below implementation you can only find the length of the longest subsequence but
not the subsequence.
*/

// the header files included are for std io , string and file stream.
#include<iostream>
#include<string>
#include<fstream>
#include<time.h>
using namespace std;

/* the class lcs that has the two functions. one will read the strings from the file and write
the lcs length.
and the other will calculate the just the lcs via naive recursive algorithm */

class LCS
{
    public:
        int rows , columns ;  // need the rows and columns for strings of different lengths
        int c [ 30 ] [ 30 ] ; // have given the maximum size of the memo as 30 since do not
think will consider strings over
                                // length considering the experience with naive recursive algorithm
                    void arr_initialise() ;  // initialiases the array with null elements
        int lcs_len( string, string, int, int ) ;
        void lcs_read_write();
};

void LCS :: arr_initialise()
{
    for ( int i = 0 ; i < 30 ; i++ )
            {
                    for ( int j = 0 ; j < 30 ; j++ )
                    {
                            c [ i ] [ j ] = NULL ;
                    }
            }
}

/* below is the function that calculates the lcs via naive recursive algorithm */
int LCS :: lcs_len( string A, string B, int i, int j )
{
    if ( ( c[ i ] [ j ] == NULL ) and ( i >= 0 ) and ( j >= 0 ) ) // taking note of the memo and
returning the same
```

```
        {
                if ( A[ i ] == B[ j ] )
                {
                        return c [ i ] [ j ] = lcs_len ( A , B , i - 1 , j - 1 ) + 1 ;
                }
                else
                {
                        return c[ i ] [ j ] = max( lcs_len( A, B, i - 1, j ) , lcs_len( A, B, i, j - 1 ) ) ;

                }
        }
        else
                return c [ i ] [ j ] ;
}

/* This is the function that will read the strings from the file and write the output to console
*/
void LCS :: lcs_read_write()
{
   string line, A, B ;
   int position, length, lcs_length, case_num = 1 ;
   clock_t t ;
   ifstream myfile ;
   myfile.open("lcs_test_case.txt", ios::in); // reading in from the file
   if ( myfile.is_open() )
   {
      while( myfile.good() )
      {
         getline( myfile, line );
         position = line.find(' ');
         length = line.length();
         A = line.substr( 0 , position ) ;
         B = line.substr( position + 1 , length ) ;
         rows = A.length() ;
         columns = B.length() ;
                        arr_initialise() ;
         t = clock() ;
         cout << " case " << case_num << " lcs length is: " << lcs_len( A, B, rows - 1 ,
columns - 1 ) ;
         cout<< " and the time taken is " << ( double ) ( ( clock() - t ) / CLOCKS_PER_SEC )
<< endl ;
         case_num++ ;
      }
      myfile.close();
   }
   else
   {
      cout << " \n Unable to open the file. The input case file may not exist. " ;
   }
}
```

```
/* The main program that will only call the read write function for LCS */
int main()
{
    LCS l ;
    l.lcs_read_write();
    return 0;
}
```

Couldn't do the analysis for the runs since it was not giving the results properly every time some thing or the other was going wrong.

Cannot say for sure as to what kind of process improvement it can give since could not run the same but can be sure that the performance will be a bit better than the naïve recursive since a memo is already made but not much of a advantage though.

Dynamic Programming :

Dynamic programming seems to be the perfect natural choice for finding the LCS. You just make a table for finding the matches and that way you get the largest number of LCS at the cell in last row and column. You then traverse the table which is usually a 2D array collecting the characters where the change takes place and thus getting the LCS. The only over head right now visible is that of the integer table which is a matrix or a 2D array. The longer your string the bigger should be the matrix.

Below is the code for the same.

```
/* This is the c++ implementation of the dynamic programming for LCS. */
/* The following code should print the LCS as well as the length of the strings. */

#include<iostream>
#include<string>
#include<fstream>
#include<time.h>

using namespace std ;

class LCS
{
    public:
        void read_write();
        void dyn_prog( string, string );
                timespec diff( timespec , timespec );
};

void LCS :: dyn_prog( string A, string B )
{
    int rows = A.length() + 1 ;
    int columns = B.length() + 1 ;
    int table [ rows ] [ columns ] ;

    // initialise the arrays
    for ( int i = 0 ; i < rows ; i++ )
    {
        for ( int j = 0 ; j < columns ; j++ )
        {
            table [ i ] [ j ] = 0 ;
        }
    }

    // make the table
    for ( int i = 1 ; i < rows ; i++ )
    {
        for ( int j = 1 ; j < columns ; j++ )
        {
            if ( A [ i - 1 ] == B [ j - 1 ] )
            {
```

```
            table [ i ] [ j ] = table [ i - 1 ] [ j - 1 ] + 1 ;
         }
         else
         {
            table [ i ] [ j ] = max( table [ i - 1 ] [ j ] , table [ i ] [ j - 1 ] ) ;
         }
      }
   }

        string lcs = "" ;
        // Traverse the table and put the characters of the LCS that match into the empty
string lcs that will be retruned or printed.
        while ( ( rows != 0 ) and ( columns != 0 ) )
        {
                if ( table [ rows - 1 ] [ columns - 1 ] == table [ rows - 1 ] [ columns - 2 ] )

                        columns-- ;
                else if ( table [ rows - 1 ] [ columns - 1 ] == table [ rows - 2 ] [ columns - 1 ] )

                        rows-- ;
                else
                {
                        lcs.push_back( A [ rows - 2 ] ) ;
                        rows-- ;
                        columns-- ;
                }
        }

   cout << " the length of lcs is " << lcs.length() << " for string length " << A.length() <<"
and the lcs is: " << " " ;
   for ( int i = lcs.length() - 1 ; i >= 0 ; i-- )
   {
      cout << lcs[ i ] ;
   }
}

void LCS :: read_write()
{
   string A , B , line ; // A, B for the strings that have LCS and line for getting the line from
file that has both the strings
   ifstream myfile ;
   int case_num = 1 ;
   struct timespec ts, te; // The structure entries for start and end time.
   long r1, r2;
   myfile.open( "dna_input_long.txt" ) ; // file opened for reading with ifstream so ios::in
mode is default
   while ( !myfile.eof() )
        {
                getline( myfile , line ) ;
                A = line.substr( 0 , line.find(' ') ) ;
                B = line.substr( line.find(' ') + 1 , line.length() );
                cout << " case " << case_num ;
```

```
                clock_gettime( CLOCK_PROCESS_CPUTIME_ID , &ts );
                dyn_prog( A, B );
                clock_gettime( CLOCK_PROCESS_CPUTIME_ID , &te );
                cout << " time taken "<< diff( ts, te ).tv_sec << ":" << diff( ts, te ).tv_nsec <<
endl ;
                case_num++;
        }
        myfile.close() ;
   myfile.open( "binary_seq_input_long.txt" ) ; // file opened for reading with ifstream so
ios::in mode is default
   while ( !myfile.eof() )
        {
                getline( myfile , line ) ;
                A = line.substr( 0 , line.find(' ') ) ;
                B = line.substr( line.find(' ') + 1 , line.length() );
                cout << "case " << case_num ;
                clock_gettime( CLOCK_PROCESS_CPUTIME_ID , &ts );
                dyn_prog( A, B );
                clock_gettime( CLOCK_PROCESS_CPUTIME_ID , &te );
                cout << " time taken " << diff( ts, te ).tv_sec << ":" << diff( ts, te ).tv_nsec <<
endl ;
                case_num++;
        }
        myfile.close() ;
}

// The following piece of code is adopted from www.guyrutenberg.com/2007/09/22profiling-
code-using-clock_gettime
timespec LCS :: diff( timespec start, timespec end )
{
        timespec temp ;
        if ( ( end.tv_nsec - start.tv_nsec ) < 0 )
        {
                temp.tv_sec = end.tv_sec - start.tv_sec - 1 ;
                temp.tv_nsec = 1000000000 + end.tv_nsec - start.tv_nsec ;
        }
        else
        {
                temp.tv_sec = end.tv_sec - start.tv_sec ;
                temp.tv_nsec = end.tv_nsec - start.tv_nsec ;
        }
        return temp ;
}

int main()
{
   LCS l ;
   l.read_write() ;
   return 0 ;
}
```

The program was run with an increased execution heap size of 5242880KB which is roughly equal to 50MB. The command for so is ulimit -s <size in KB> in linux. Note that the program references libraries of time and they have to linked while making the output file. So when you make the output file the command is g++ -lrt -o lcs_dynamic_programming.out lcs_dynamic_programming.cpp . The resultant program when ran on terminal can flood the same so you should redirect the output to another file like ./lcs_dynamic_programming.out > dynamic_long_output.txt

Results of the run:

The dynamic programming required loads of memory or heap execution space. As the size of the strings that were being tested increased so had the heap size had to be increased. In this case for each of the two strings being 28000 characters long the heap space had to be about 5242880KB or 50 MB long. For strings longer than that like even for strings like 30,000 characters long each even heap of 1GB proved inadequate or gave a segmentation error. Looks like the table which is a matrix requires a lot of calculations.

Thus this algorithm is quadratic in space and time but only by a square magnitude. Thus for lower times you get the solution but you have to sacrifice a lot of memory for the execution almost square to order of the strings that are in question.

Also one more thing is that Dynamic Programming takes almost the same execution time for good case and worst case. On the whole we can take a average case running time for this algorithm.

The results for the dynamic programming for LCS are on the next page with problem sizes and running times.

String length of each of the strings	LCS length for the DNA sequence	Time taken for DNA sequence in seconds	LCS length for the binary sequence	Time taken for binary sequence in seconds
5000	3267	0.833703247	4057	0.744986917
5500	3584	0.932162015	4457	0.911630677
6000	3909	1.487607714	4852	1.389514067
6500	4243	1.296524137	5263	1.251276906
7000	4576	1.499243271	5669	1.450382615
7500	4900	1.722097574	6099	1.662104636
8000	5236	1.960009042	6484	1.888218899
8500	5540	2.21191427	6870	2.127492761
9000	5848	2.474060494	7294	2.396453257
9500	6195	2.759068236	7690	2.66612004
10000	6490	3.60019295	8108	2.953758227
10500	6833	3.364206593	8510	3.258880727
11000	7179	3.681984542	8920	3.57980529
11500	7489	4.3499519	9298	3.896376909
12000	7817	4.396290684	9715	4.253355737
12500	8167	4.770638194	10123	4.614469928
13000	8472	5.152305904	10545	4.981606861
13500	8823	5.56992761	10960	5.386884697
14000	9142	5.982990312	11365	5.77456582
14500	9472	6.410588531	11763	6.214341311
15000	9777	6.848767898	12181	6.65929974
15500	10125	7.323251122	12536	7.83894607
16000	10436	7.790638939	12964	7.553836007
16500	10745	8.287762504	13380	8.50091162
17000	11100	8.811743396	13798	8.626162763
17500	11420	9.318655962	14185	9.49597302
18000	11769	9.855419848	14602	9.567704323
18500	12075	10.411791418	14973	10.124145446
19000	12411	11.20962804	15424	10.673306632
19500	12730	11.814407759	15796	11.362966519
20000	13083	12.205868785	16257	11.851685181
20500	13384	12.786892512	16615	12.432365733
21000	13726	13.411778427	17024	13.4162456
21500	14062	14.45644025	17472	13.679488031
22000	14362	14.726880551	17852	14.297610049
22500	14676	15.40358393	18239	14.941842579
23000	15025	16.85113823	18676	15.666495625
23500	15403	16.79278734	19077	16.307396684
24000	15678	17.58204583	19464	17.6003173
24500	16007	18.257118653	19886	17.805919966
25000	16342	19.3236082	20265	18.50931609
25500	16646	19.847946589	20679	19.493423679
26000	16990	20.568325081	21110	20.85861358
26500	17301	21.302084856	21489	20.928867781
27000	17653	22.11803363	21898	21.600190718
27500	17979	22.955870436	22339	22.384705039

Hirschberg:

Hirschberg algorithm for LCS is quadratic in time since it is recursive but it is linear in space. This is because it uses divide and conquer methodology and recursively looks into only the smaller sub problems that could give the solution. And at any time it is only looking at the maximum length of LCS and not at the entire table which allows it to achieve this. Following is the code.

```
/* Program to implement the largest common subsequence with help of Hirschberg
algorithm. */

#include<iostream>
#include<string>
#include<fstream>
#include<time.h>

using namespace std ;

/* the class has methods for read write algorithms B and algorithm C and profiling
timespec */
class Hirschberg
{
        public:
                void B( int, int, string, string, int[] );
                string C( int, int, string, string, string );
                void read_write();
                timespec diff( timespec, timespec ) ;
};

/* Algorithm B that delivers only the last row in the matrix. Since matrix passed by
reference no need to pass the matrix back */
void Hirschberg :: B( int m, int n, string str1, string str2, int LL [] )
{
        int k [2] [ n + 1 ] ;
    int i, j;
    for ( j = 0 ; j < ( n + 1 ); j++ )
                k [1] [j] = 0 ;

        for ( i = 0 ; i < m ; i++ )
        {
                for ( j = 0 ; j < ( n + 1 ) ; j++ )
                        k [0] [j] = k [1] [j] ;

                for ( j = 1 ; j < ( n + 1 ) ; j++ )
                {
                        if ( str1[ i ] == str2[ j - 1 ] )
                        {
                                k [1] [j] = k [0] [j-1] + 1 ;
                        }
                        else
                        {
                                k [1] [j] = max( k [1] [j-1] , k [0] [j] ) ;
```

```
                        }
                }
        }
    for ( j = 0 ; j < ( n + 1 ); j++ )
                LL [j] = k [1] [j] ;
}

/* The program that writes the LCS string has to return since it is used in recursive fashion.
*/
string Hirschberg :: C( int m, int n, string str1, string str2, string lcs )
{
        // If problem is trivial then solve it.
        if( n == 0 )
        {
                lcs = "" ;
                return lcs ;
        }
        else
        {
                if ( m == 1 )
                {
                        int found ;
                        found = str2.find_first_of( str1[0] );
                        if ( found != string::npos )
                                lcs = str1[ 0 ] ;
                        else
                                lcs = "" ;
                                return lcs ;
                }
        }

        int l1[ n + 1 ], l2[ n + 1 ] ;
        int i = ( int ) ( m / 2 ) ;
        B( i, n, str1.substr( 0, i ), str2, l1 ) ;

        // The statements that reverse the string with the helper functions like rbegin, rend
and riterator in string class.
        string str1_rev, str2_rev ;
        string::reverse_iterator rev_iter ;
        for ( rev_iter = str1.rbegin() ; rev_iter < str1.rend() ; rev_iter++ )
                str1_rev.push_back( *rev_iter );
        for ( rev_iter = str2.rbegin() ; rev_iter < str2.rend() ; rev_iter++ )
                str2_rev.push_back( *rev_iter );
    B( i + 1 , n, str1_rev.substr( 0, m - i ), str2_rev, l2 );

    // Finding the max value of j.
        int max = 0, k = 0 ;
        for ( int j = 0 ; j < ( n + 1 ) ; j++ )
        {
                if ( ( l1[ j ] + l2[ n - j ] ) > max )
                {
                        max = l1[ j ] + l2[ n - j ] ;
```

```
                        k = j ;
                }
        }
        string lcs1, lcs2 ;

        /* Have to increase the value of i in case it becomes zero since then the C++
substring function will
        send back a empty string and not the first character. */
        if ( i == 0 ) i++ ;
        return C( i, k, str1.substr( 0, i ), str2.substr( 0, k ), lcs1 ).append( C( m - i , n - k ,
str1.substr( i, m ), str2.substr( k, n ), lcs2 ) ) ;
}

// the following piece of code is from www.guyrutenberg.com/2007/09/22profiling-code-
using-clock_gettime
timespec Hirschberg :: diff ( timespec start , timespec end )
{
        timespec temp ;
        if ( ( end.tv_nsec - start.tv_nsec ) < 0 )
        {
                temp.tv_sec = end.tv_sec - start.tv_sec + 1 ;
                temp.tv_nsec = 1000000000 + end.tv_nsec - start.tv_nsec ;
        }
        else
        {
                temp.tv_sec = end.tv_sec - start.tv_sec ;
                temp.tv_nsec = end.tv_nsec - start.tv_nsec ;
        }
        return temp ;
}

void Hirschberg :: read_write()
{
        string s1, s2, s3, line ;
        ifstream myfile ;
        int case_num = 1 ;
        struct timespec ts, te ;
   myfile.open("dna_input_long.txt");
   while( !myfile.eof() )
        {
                getline( myfile, line ) ;
                s1 = line.substr( 0 , line.find(' ') ) ;
                s2 = line.substr( line.find(' ') + 1 , line.length() ) ;
           int len1 = s1.length() , len2 = s2.length() ;
                clock_gettime( CLOCK_PROCESS_CPUTIME_ID , &ts ) ;
                string s4 = C ( len1, len2, s1, s2, s3 ) ;
                clock_gettime( CLOCK_PROCESS_CPUTIME_ID , &te ) ;
                cout << "case " << case_num << " for string length " << len1 << " " << " lcs
length is "<< s4.length() << " and lcs is " << s4 << " time taken " << diff( ts, te ).tv_sec <<
":" << diff( ts, te).tv_nsec << endl ;
                case_num++ ;
        }
```

```
    myfile.close() ;
    myfile.open("binary_seq_input_long.txt");
    while( !myfile.eof() )
          {
                getline( myfile, line ) ;
                s1 = line.substr( 0 , line.find(' ') ) ;
                s2 = line.substr( line.find(' ') + 1 , line.length() ) ;
             int len1 = s1.length() , len2 = s2.length() ;
                clock_gettime( CLOCK_PROCESS_CPUTIME_ID , &ts ) ;
                string s4 = C ( len1, len2, s1, s2, s3 ) ;
                clock_gettime( CLOCK_PROCESS_CPUTIME_ID , &te ) ;
                cout<< "case " << case_num << " for string length " << len1 << " lcs length is
"<< s4.length() << " and lcs is " << s4 << " time taken " << diff( ts, te ).tv_sec << ":" <<
diff( ts, te).tv_nsec << endl ;
                case_num++ ;
          }
    myfile.close() ;
}

int main()
{
    Hirschberg h ;
    h.read_write() ;
    return 0;
}
```

When we ran Hirschberg on large data on DNA sequence of up to lengths of string 200,000 each it took around 2,316 seconds. Below is the table given of the performance.

String length of each string	Length of the LCS string	Time required in seconds
60000	39245	207.356579829
80000	52282	370.454002464
100000	65367	596.314238720
120000	78465	890.133057993
140000	91605	1186.410035870
160000	104675	1497.588193529
180000	117768	1870.844664669
200000	130821	2316.805038850

On the next page is the table of the results using the same input data that was used in the previous dynamic programming case. But with one catch while dynamic programming needs execution heap size of around 50MB while the Hirschberg only required 8KB memory but only required roughly twice the time as opposed to dynamic programming.

String length of each of the strings	LCS length for the DNA sequence	Time taken for DNA sequence in seconds	LCS length for the binary sequence	Time taken for binary sequence in seconds
5000	3267	1.478500025	4057	1.425933669
5500	3584	3.772585985	4457	3.713959871
6000	3909	2.92639253	4852	2.3149881
6500	4243	2.458599175	5263	2.38813927
7000	4576	4.880687468	5669	4.76466672
7500	4900	3.254281572	6099	3.166603071
8000	5236	5.707707348	6484	5.604751588
8500	5540	4.18243294	6870	4.73731179
9000	5848	6.701840598	7294	6.580322683
9500	6195	5.226042885	7690	5.7079105
10000	6490	7.779491858	8108	5.742579843
10500	6833	6.371580973	8510	8.204318156
11000	7179	8.978755599	8920	6.79055808
11500	7489	9.634619449	9298	9.418523135
12000	7817	10.691833891	9715	8.561356111
12500	8167	9.9879439	10123	10.76906178
13000	8472	9.735108001	10545	11.483770162
13500	8823	12.516096959	10960	10.227146914
14000	9142	11.298179898	11365	12.980120371
14500	9472	12.109793563	11763	12.237127761
15000	9777	14.955321876	12181	14.610024573
15500	10125	15.852458307	12536	13.494113439
16000	10436	16.746458447	12964	14.321677484
16500	10745	17.669384294	13380	17.324388059
17000	11100	16.64337399	13798	16.288218899
17500	11420	19.932081487	14185	19.452290596
18000	11769	20.713858391	14602	18.198239507
18500	12075	21.731793065	14973	19.136117121
19000	12411	20.771956769	15424	20.231191519
19500	12730	23.905442583	15796	21.425130146
20000	13083	25.256589436	16257	24.427772279
20500	13384	24.385563872	16615	25.591027093
21000	13726	25.379459512	17024	24.669698464
21500	14062	28.594428501	17472	27.897640977
22000	14362	28.410075564	17852	27.277816133
22500	14676	31.236177142	18239	30.59751067
23000	15025	30.440830445	18676	30.34098014
23500	15403	32.73972232	19077	33.711770912
24000	15678	33.175052325	19464	32.370123603
24500	16007	36.882325332	19886	35.722712231
25000	16342	36.69608729	20265	36.913173053
25500	16646	39.519525569	20679	36.381342897
26000	16990	40.905600246	21110	39.85481553
26500	17301	40.753439694	21489	41.816917722
27000	17653	43.824624275	21898	40.59910382
27500	17979	45.560077132	22339	44.266352911

A sample run of Hirschberg for string length 20000.

The program required 33:539587755 seconds to get the LCS which was of length 13064. A sample gprof run is given below to get the connection between the stack calls and time spent.

Flat profile:
Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
80.81	14.87	14.87	37420	0.00	0.00	Hirschberg::B(int, int, std::string, std::string, int*)
17.31	18.05	3.18	600117922	0.00	0.00	int const& std::max<int>(int const&, int const&)
1.47	18.32	0.27	1	0.27	0.27	global constructors keyed to Hirschberg::B(int, int, std::string, std::string, int*)
0.11	18.34	0.02	1218004	0.00	0.00	std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> >::base() const
0.11	18.36	0.02	609002	0.00	0.00	bool __gnu_cxx::operator< <char*, std::string>(__gnu_cxx::__normal_iterator<char*, std::string> const&, __gnu_cxx::__normal_iterator<char*, std::string> const&)
0.11	18.38	0.02	571582	0.00	0.00	std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> >::reverse_iterator(std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> > const&)
0.05	18.39	0.01	1143164	0.00	0.00	__gnu_cxx::__normal_iterator<char*, std::string>::operator--()
0.05	18.40	0.01	609002	0.00	0.00	bool std::operator< <__gnu_cxx::__normal_iterator<char*, std::string> >(std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> > const&, std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> > const&)
0.05	18.41	0.01	571582	0.00	0.00	__gnu_cxx::__normal_iterator<char*, std::string>::operator*() const
0.05	18.42	0.01	2	0.01	9.08	Hirschberg::C(int, int, std::string, std::string, std::string)
0.00	18.42	0.00	1218004	0.00	0.00	__gnu_cxx::__normal_iterator<char*, std::string>::base() const
0.00	18.42	0.00	590292	0.00	0.00	std::iterator<std::random_access_iterator_tag, char, long, char*, char&>::iterator()
0.00	18.42	0.00	571582	0.00	0.00	std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> >::operator*() const
0.00	18.42	0.00	571582	0.00	0.00	std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> >::operator++(int)
0.00	18.42	0.00	18710	0.00	0.00	__gnu_cxx::__normal_iterator<char*, std::string>::__normal_iterator()
0.00	18.42	0.00	18710	0.00	0.00	std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> >::reverse_iterator()
0.00	18.42	0.00	4	0.00	0.00	Hirschberg::diff(timespec, timespec)
0.00	18.42	0.00	1	0.00	0.00	__static_initialization_and_destruction_0(int, int)
0.00	18.42	0.00	1	0.00	18.15	Hirschberg::read_write()

granularity: each sample hit covers 2 byte(s) for 0.05% of 18.42 seconds

index % time self children called name
<spontaneous>
[1] 98.5 0.00 18.15 main [1]
0.00 18.15 1/1 Hirschberg::read_write() [3]

37420 Hirschberg::C(int, int, std::string, std::string, std::string) [2]
0.01 18.14 2/2 Hirschberg::read_write() [3]
[2] 98.5 0.01 18.14 2+37420 Hirschberg::C(int, int, std::string, std::string, std::string) [2]
14.87 3.18 37420/37420 Hirschberg::B(int, int, std::string, std::string, int*) [4]
0.01 0.04 609002/609002 bool std::operator<<__gnu_cxx::__normal_iterator<char*, std::string> >(std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> > const&, std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> > const&) [8]
0.00 0.03 571582/571582 std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> >::operator++(int) [9]
0.00 0.02 571582/571582 std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> >::operator*() const [13]
0.00 0.00 18710/18710 std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> >::reverse_iterator() [23]
37420 Hirschberg::C(int, int, std::string, std::string, std::string) [2]

0.00 18.15 1/1 main [1]
[3] 98.5 0.00 18.15 1 Hirschberg::read_write() [3]
0.01 18.14 2/2 Hirschberg::C(int, int, std::string, std::string, std::string) [2]
0.00 0.00 4/4 Hirschberg::diff(timespec, timespec) [24]

14.87 3.18 37420/37420 Hirschberg::C(int, int, std::string, std::string, std::string) [2]
[4] 98.0 14.87 3.18 37420 Hirschberg::B(int, int, std::string, std::string, int*) [4]
3.18 0.00 600117922/600117922 int const& std::max<int>(int const&, int const&) [5]

3.18 0.00 600117922/600117922 Hirschberg::B(int, int, std::string, std::string, int*) [4]
[5] 17.3 3.18 0.00 600117922 int const& std::max<int>(int const&, int const&) [5]

0.27 0.00 1/1 __do_global_ctors_aux [7]
[6] 1.5 0.27 0.00 1 global constructors keyed to Hirschberg::B(int, int, std::string, std::string, int*) [6]
0.00 0.00 1/1 __static_initialization_and_destruction_0(int, int) [25]

<spontaneous>
[7] 1.5 0.00 0.27 __do_global_ctors_aux [7]
0.27 0.00 1/1 global constructors keyed to Hirschberg::B(int, int, std::string, std::string, int*) [6]

0.01 0.04 609002/609002 Hirschberg::C(int, int, std::string, std::string, std::string) [2]
[8] 0.3 0.01 0.04 609002 bool std::operator< <__gnu_cxx::__normal_iterator<char*, std::string> >(std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> > const&, std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> > const&) [8]
0.02 0.00 1218004/1218004 std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> >::base() const [10]
0.02 0.00 609002/609002 bool __gnu_cxx::operator< <char*, std::string>(__gnu_cxx::__normal_iterator<char*, std::string> const&, __gnu_cxx::__normal_iterator<char*, std::string> const&) [11]

0.00 0.03 571582/571582 Hirschberg::C(int, int, std::string, std::string, std::string) [2]
[9] 0.1 0.00 0.03 571582 std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> >::operator++(int) [9]
0.02 0.00 571582/571582 std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> >::reverse_iterator(std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> > const&) [12]
0.01 0.00 571582/1143164 __gnu_cxx::__normal_iterator<char*, std::string>::operator--() [14]

0.02 0.00 1218004/1218004 bool std::operator< <__gnu_cxx::__normal_iterator<char*, std::string> >(std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> > const&, std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> > const&) [8]
[10] 0.1 0.02 0.00 1218004 std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> >::base() const [10]

0.02 0.00 609002/609002 bool std::operator< <__gnu_cxx::__normal_iterator<char*, std::string> >(std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> > const&, std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> > const&) [8]
[11] 0.1 0.02 0.00 609002 bool __gnu_cxx::operator< <char*, std::string>(__gnu_cxx::__normal_iterator<char*, std::string> const&, __gnu_cxx::__normal_iterator<char*, std::string> const&) [11]
0.00 0.00 1218004/1218004 __gnu_cxx::__normal_iterator<char*, std::string>::base() const [20]

0.02 0.00 571582/571582 std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> >::operator++(int) [9]
[12] 0.1 0.02 0.00 571582 std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> >::reverse_iterator(std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string>

> const&) [12]
0.00 0.00 571582/590292 std::iterator<std::random_access_iterator_tag, char, long, char*, char&>::iterator() [21]

0.00 0.02 571582/571582 Hirschberg::C(int, int, std::string, std::string, std::string) [2]
[13] 0.1 0.00 0.02 571582 std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> >::operator*() const [13]
0.01 0.00 571582/571582 __gnu_cxx::__normal_iterator<char*, std::string>::operator*() const [15]
0.01 0.00 571582/1143164 __gnu_cxx::__normal_iterator<char*, std::string>::operator--() [14]

0.01 0.00 571582/1143164 std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> >::operator++(int) [9]
0.01 0.00 571582/1143164 std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> >::operator*() const [13]
[14] 0.1 0.01 0.00 1143164 __gnu_cxx::__normal_iterator<char*, std::string>::operator--() [14]

0.01 0.00 571582/571582 std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> >::operator*() const [13]
[15] 0.1 0.01 0.00 571582 __gnu_cxx::__normal_iterator<char*, std::string>::operator*() const [15]

0.00 0.00 1218004/1218004 bool __gnu_cxx::operator< <char*, std::string>(__gnu_cxx::__normal_iterator<char*, std::string> const&, __gnu_cxx::__normal_iterator<char*, std::string> const&) [11]
[20] 0.0 0.00 0.00 1218004 __gnu_cxx::__normal_iterator<char*, std::string>::base() const [20]

0.00 0.00 18710/590292 std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> >::reverse_iterator() [23]
0.00 0.00 571582/590292 std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> >::reverse_iterator(std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> > const&) [12]
[21] 0.0 0.00 0.00 590292 std::iterator<std::random_access_iterator_tag, char, long, char*, char&>::iterator() [21]

0.00 0.00 18710/18710 std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> >::reverse_iterator() [23]
[22] 0.0 0.00 0.00 18710 __gnu_cxx::__normal_iterator<char*, std::string>::__normal_iterator() [22]

0.00 0.00 18710/18710 Hirschberg::C(int, int, std::string, std::string,

std::string) [2]
[23] 0.0 0.00 0.00 18710 std::reverse_iterator<__gnu_cxx::__normal_iterator<char*, std::string> >::reverse_iterator() [23]
0.00 0.00 18710/590292 std::iterator<std::random_access_iterator_tag, char, long, char*, char&>::iterator() [21]
0.00 0.00 18710/18710 __gnu_cxx::__normal_iterator<char*, std::string>::__normal_iterator() [22]

0.00 0.00 4/4 Hirschberg::read_write() [3]
[24] 0.0 0.00 0.00 4 Hirschberg::diff(timespec, timespec) [24]

0.00 0.00 1/1 global constructors keyed to Hirschberg::B(int, int, std::string, std::string, int*) [6]
[25] 0.0 0.00 0.00 1 __static_initialization_and_destruction_0(int, int) [25]

The statistics show an important data that though the recursive calls to algorithm C in Hirschberg outstripped the calls to algorithm B by an order of roughly around 16000. But most of the time of the program was spent in algorithm B that is calculating the matrix and getting the lengths of sub sequences and the split partition indexes. Almost 80% of the time is consumed by Algorithm B in Hirschberg case.

Solving Problems from Page 396 / 397 of textbook

Ex 15.4.1
The LCS of <1,0,0,1,0,1,0,1> and <0,1,0,1,1,0,1,1,0> is 010101.

Ex 15.4.2
Pseudocode to reconstruct LCS from completed c table without using b table. And only from the sequence X = < x1, x2, …,xm> in O(m + n) time without using table b.

```
declare lcs → "null string"
while( m != 0 and n!= 0 )
   if c [ m – 1 ] [ n – 1 ] = c [ m – 1 ] [ n – 2 ]
      m → m – 1
   else if c [ m – 1 ] [ n – 1 ] = c [ m – 2  ] [ n – 1 ]
      n → n – 1
   else
       append to string lcs ; the char from sequence X< x : m – 2 >
       m → m – 1
       n → n – 1
end
```

Ex 15.4.4
You only need a table of mxn entries if you want to reconstruct the LCS. If you only want the length of the LCS then you are only interested in the cell that is in last row and last column. To reconstruct the same you only need to know the row above the one that you are at present calculating. This thing can be easily achieved by taking the matrix column length as min(m, n) . In this case let us assume the minimum is n then a matrix of 2 rows and n columns should do the trick and applying algorithm B from the Hirschberg should be able to give us the matrix we need. We just look at the last entry and that gives us the longest subsequence length.
Pseudo code.

```
Initialise matrix K( 1, j ) ← 0 [ j = 0,...,n]
for I ← 1 to m do
   K (0 , j ) ← K( 1, j ) [ j = 0,...n ]
    for j ← 1 to n do
       if A( I ) = B( j ) then K( 1, j ) ← K ( 0, j – 1 ) + 1
          else K( 1, j ) ← max{ K( 1, j – 1 ), K ( 0, j ) } ;
end
length of subsequence ← K [ 1 ] [ n ]
```

Ex 15.4.5
Find largest monotonically increasing sub sequences of a sequence of n numbers.
To do so first sort the sequence of n numbers by say quick sort algorithm which has complexity of n lg n and store the sorted array as a separate string sequence leaving the original sequence untouched.
Now the problem is to find the LCS from the sorted sequence in the unsorted sequence which will give us the monotonically increasing sub sequence of the numbers from the original sequence. A dynamic programming algorithm can do this in O(n square) n being the length of the sequence.

Hardware and Platform:

All the experiments were ran on a machine that has intel core 2 duo with clock speed of 2.1GHz and 2MB L2 cache. The machine has 4GB of primary memory. The implementation language for all the programs was C++ and was compiled with g++. The operating system was ubuntu 10.04.
All the programs were ran on the execution heap size of 8KB even the large ones for Hirschberg algorithm implementation. However for the dynamic programming had to increase the heap memory execution size to 5242880 KB and it ran pretty well up to strings each of length 28000.