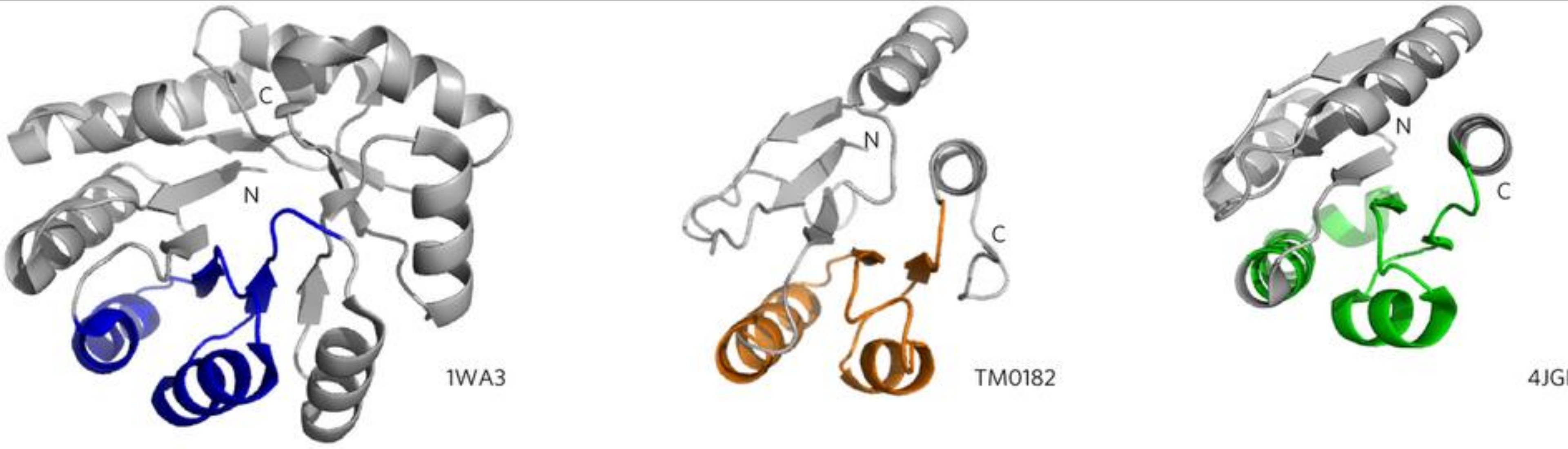
Longest Common Subsequence



```
1WA3      11 KIVAVLR---ANsveeak------------------------ekalaVF----EGGv---HLIEITFTvP-   46
              ++                                                         ++  +|
TM0182     1 MYILFRE---MK-NNWY--SLAALLSTiysrhldVEARPV----KFEEI----KKFPpeKTIVAYSFMSF-   56
               |+                   +  +        |+  +        |    |       ||  | +
4JGI      90 AKIVLATvegDLhDIGKniFRTMAEASg------FEVFDLgidvPVKIIvdkvKEVN--PEIVGLSGVLTl   152


1WA3      47 DADTVIKELSFLKEK-GA---IIGAGT--VTsveqCr-kAVE-SGAEFIVSpHldeeisqfckekgVFYMPG  110
             |+|||  |+  |||+ |     + ||   ||         +   | + + +   +| |  |          |
TM0182    57 DLDTVREEVKTLKER-GY---TLIAGGphVTa---DpegCLR-MGFDHVFTgDGeENILKFLMGErKKIFDG  120
              ||++|| |  ||         +|+||  |             +| |   |       +|
4JGI     153 ALDSMRETVDALKAEgLRndlKVIIGGvpVNe---N---VCQrVGADDFST-NA-ADGVKICQRW-vg----  211
```

The Unix *diff* command compares two text files and provides a report on their differences.

It works line by line, and reports the individual lines that must be added to or deleted from *file1* to obtain *file2*

```
a.txt
a
b
c
d
e
f
g
```

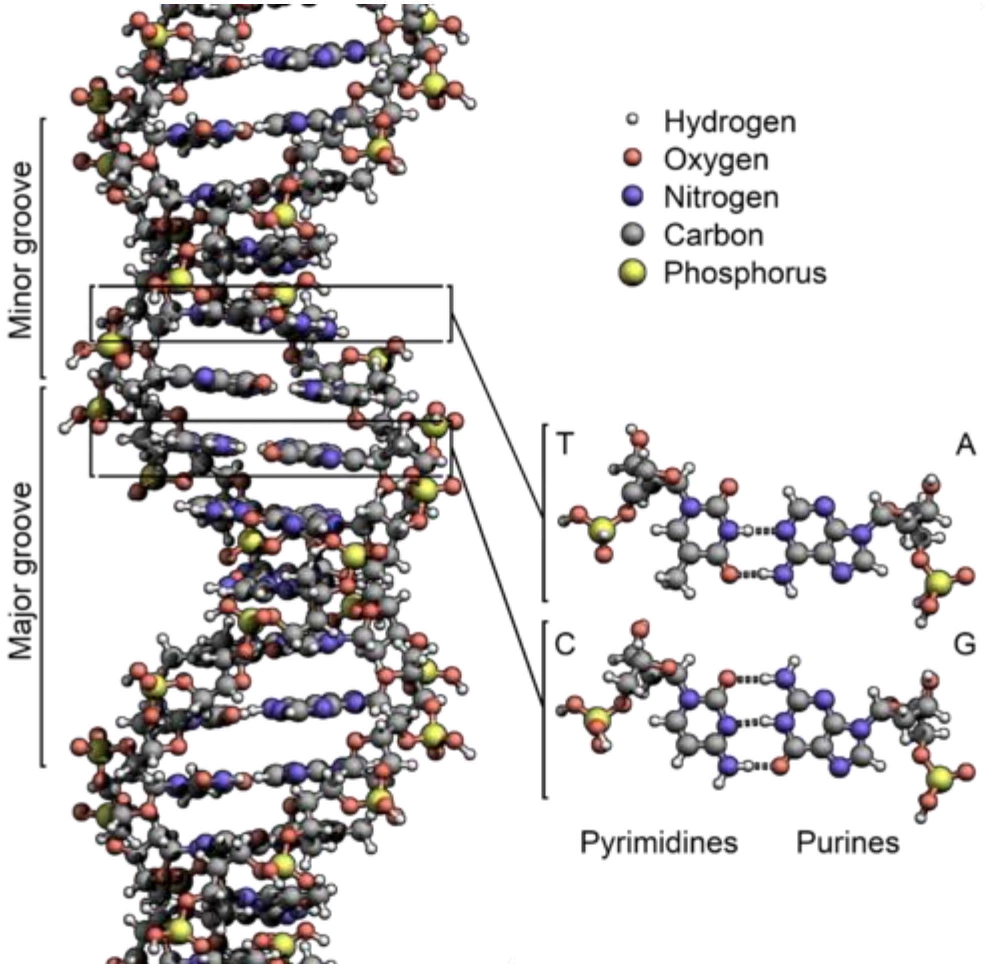
```
b.txt
a
b extra
c
d
f
new
g
new
```

```
csgate> diff a.txt b.txt
2c2
< b
---
> b extra
5d4
< e
6a6
> new
7a8
> new
csgate>
```

DNA can be represented as sequences of nucleotide bases C,G,T and A

Two organisms that have a common ancestor will have DNA sequences that are similar. The differences between them can be represented as a set of base insertions into and deletions from one sequence to obtain the other.

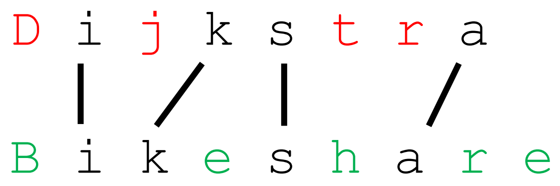
The number of changes is proportional to the number of generations since the common ancestor.



We will define the *distance* between two strings as the minimal number of additions or deletions that transform the first string into the second.

```
Dijkstra
Dijkstra
ijkstra
ikstra
iksra
iksa
Biksa
Bikesa
Bikesha
Bikeshar
Bikeshare
Bikeshare
```

9 changes



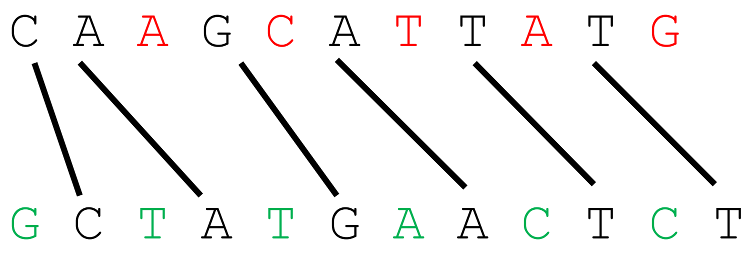
4 letters in each word don't change
8 letters in total don't change

17 letters in the original strings
17 – 9 = 8

Since we only delete or insert, the connecting lines cannot cross

The minimum number of changes is not always obvious

```
C A A G C A T T A T G
C A A G C A T T A T G
C A G C A T T A T G
C A G A T T A T G
C A G A T A T G
C A G A T T G
C A G A T T
G C A G A T T
G C T A G A T T
G C T A T G A T T
G C T A T G A A T T
G C T A T G A A C T T
G C T A T G A A C T C T
G C T A T G A A C T C T
```



The characters we identify as being unchanged are called
a *common subsequence*.

We want to identify the length of the *longest common subsequence.*

Formally, if $x_0x_1x_2x_3...x_{n-1}$ is a sequence of characters,
then a subsequence is $x_{j0}x_{j1}x_{j2}...x_{jp}$ such that
$0 \leq j_0 < j_1 < j_2 < ... < j_p \leq n-1$

Given two sequences *s* and *t*, a common subsequence is
a string *w* such that w is a subsequence of both *s* and *t*

The longest common subsequence is then the longest possible *w*.

Let *s* and *t* be two strings, let λ be the length of their longest common subsequence, and let α be any common subsequence with length λ.
Now add the same character x onto the end of both *s* and *t* to get *s*x and *t*x.
The longest common subsequence of *s*x and *t*x has length $\lambda^+ = \lambda+1$.

(1)

Proof:
αx must be a common subsequence of *s*x and *t*x, and its length is $\lambda+1$.
So $\lambda^+ \geq \lambda+1$.

Suppose $\lambda^+ > \lambda+1$.
Let β be any common subsequence of *s*x and *t*x with length λ^+.
Suppose β does not use either of the xs from *s*x and *t*x. Then we could add x onto the end of β to get a longer common subsequence. Contradiction, since λ^+ is the length of the longest common subsequence.
Therefore β must use at least one of the xs as its last character. Remove both xs from the ends of *s*x and *t*x, and so remove the last character from β to get β^-. This only reduces the length of β by 1, so $|\beta^-| > \lambda$. But β^- must be a common subsequence of *s* and *t*. Contradiction, since λ was the length of the longest common subsequence of *s* and *t*.
Therefore λ^+ cannot be $> \lambda+1$, and so $\lambda^+ = \lambda+1$

Let *s* and *t* be two strings, and let lcs(*s*,*t*) be the length of the longest common subsequence of *s* and *t.*
Now add different characters x and y onto the end of *s* and *t* to get *s*x and *t*y.
Then lcs(*s*x,*t*y) = max(lcs(*s*,*t*y), lcs(*s*x,t)).

Proof: Let β be any common subsequence of *s*x and *t*y with length lcs(*s*x,*t*y).
Suppose *β* does not use x from *s*x and does not use y from *t*y.
Then |β| = lcs(*s*,*t*) = lcs(*s*x,*t*) = lcs(*s*,*t*y)= max(lcs(*s*,*t*y), lcs(*s*x,t))

Suppose β uses the added x as its last character.
Then |β| = lcs(*s*x,t).
lcs(*s*,*t*y) cannot be bigger than β, since any common subsequence α of *s* and *t*y is also a common subsequence of *s*x and *t*y, and β was the longest.
Therefore |β| = max(lcs(*s*,*t*y), lcs(*s*x,t)).

The same argument works for β using y, and β cannot use both x and y.
Therefore results is true.

If *s* is a sequence of length n, and *t* is a sequence of length m, then we can define (for any j and k where $0 \leq j \leq n$ and $0 \leq k \leq m$):

L_{jk} to be the length of the longest common subsequence of s[0:j] and t[0:k]

If j = 0 or k = 0, then $L_{jk} = 0$ (since one of the sequences is empty).

The two rules then become:

(1) if s[j] == t[k] then $L_{j+1\ k+1} = 1 + L_{jk}$

(2) if s[j] =\= t[k] then $L_{j+1\ k+1} = \max(L_{j+1\ k}, L_{j\ k+1})$

Therefore $L_{j+1\ k+1}$ is completely determined by s[j], t[k], L_{jk}, $L_{j+1\ k}$ and $L_{j\ k+1}$

We can use these rules to create an efficient algorithm to compute L_{jk} for any j and k.

$L_{n\ m}$ is the length of the longest common subsequence of *s* and *t*

$$\begin{tikzcd}
L_{00} \arrow[dr] & L_{01} \arrow[d] \arrow[dr] & L_{02} \arrow[d] \arrow[dr] & L_{03} \arrow[d] & \cdots & & \text{all} = 0 \arrow[l] \\
L_{10} \arrow[r] \arrow[dr] & L_{11} \arrow[r] \arrow[d] & L_{12} \arrow[r] & L_{13} & & & \\
L_{20} \arrow[r] & L_{21} & \text{etc} & & & & \\
L_{30} & & & & & & \\
L_{40} & & & & & & \\
\cdots & & & & & & \\
& & & & & & \\
\text{all} = 0 \arrow[u] & & & & & &
\end{tikzcd}$$

```
def lcs(str1, str2):
    n = len(str1)
    m = len(str2)
    L = [[0] * (m+1) for i in range(n+1)]

    for j in range(n):
        for k in range(m):
            if str1[j] == str2[k]:
                L[j+1][k+1] = L[j][k] + 1
            else:
                L[j+1][k+1] = max(L[j][k+1], L[j+1][k])
    return L,n,m
```

Complexity?   Two nested loops, O(1) operations inside. So O(n*m)

1	2	3	4	5	6	7	8	9	10	11	12
C	A	A	G	G	A	T	T	A	T	G	
G	C	T	T	T	G	T	A	T	T	C	T

How do we find a common subsequence with the right length?

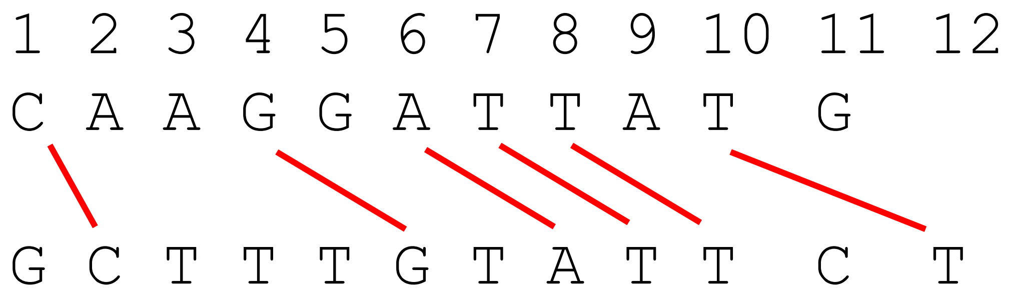
A diagonal increase indicates the use of two matching characters:

p p
p p+1

Start in bottom right corner, and move up or left until we reach [0,0].

Only move to a lower value if it is a diagonal as above. Otherwise, prefer to move up. The row and col of the diagonal you move to is a matched pair.

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1	1	1	1	1
2	0	0	1	1	1	1	1	1	2	2	2	2	2
3	0	0	1	1	1	1	1	1	2	2	2	2	2
4	0	1	1	1	1	1	2	2	2	2	2	2	2
5	0	1	1	1	1	1	2	2	2	2	2	2	2
6	0	1	1	1	1	1	2	2	3	3	3	3	3
7	0	1	1	2	2	2	2	3	3	4	4	4	4
8	0	1	1	2	3	3	3	3	3	4	5	5	5
9	0	1	1	2	3	3	3	3	4	4	5	5	5
10	0	1	1	2	3	4	4	4	4	5	5	5	6
11	0	1	1	2	3	4	5	5	5	5	5	5	6



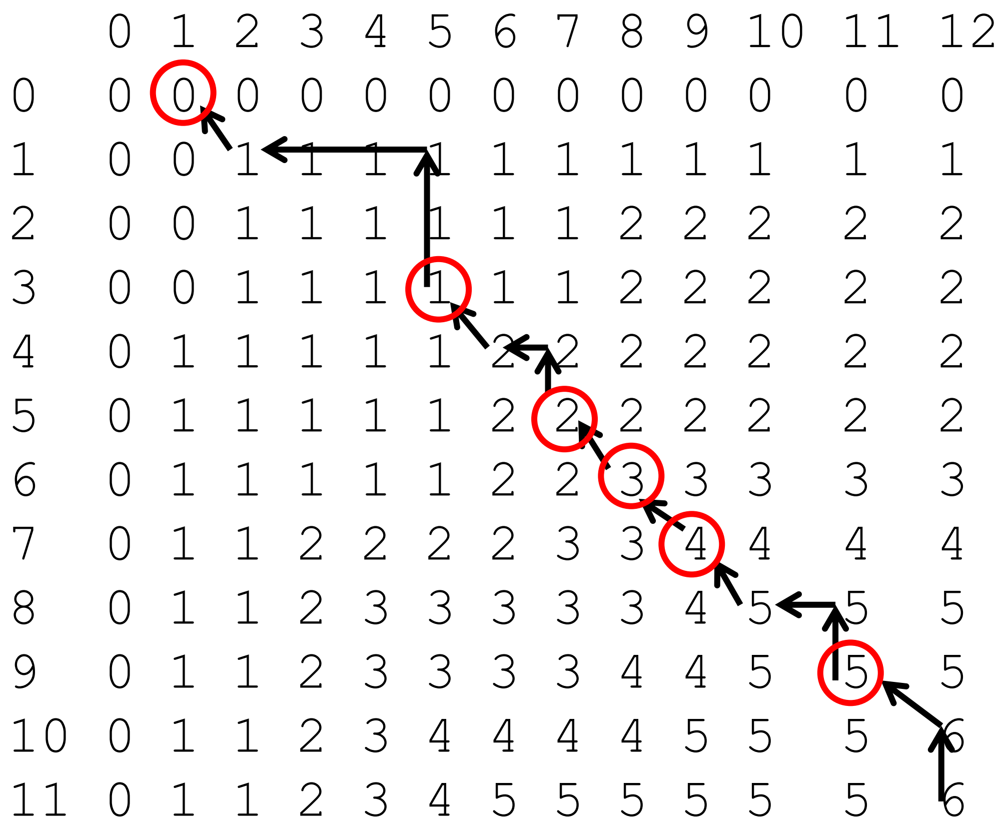
How do we find a common subsequence with the right length?

A diagonal increase indicates the use of two matching characters:

p p
p p+1

Start in bottom right corner, and move up or left until we reach [0,0].

Only move to a lower value if it is a diagonal as above. Otherwise, prefer to move up. The row and col of the diagonal you move to is a matched pair.



	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1	1	1	1	1
2	0	0	1	1	1	1	1	1	2	2	2	2	2
3	0	0	1	1	1	1	1	1	2	2	2	2	2
4	0	1	1	1	1	1	2	2	2	2	2	2	2
5	0	1	1	1	1	1	2	2	2	2	2	2	2
6	0	1	1	1	1	1	2	2	3	3	3	3	3
7	0	1	1	2	2	2	2	3	3	4	4	4	4
8	0	1	1	2	3	3	3	3	3	4	5	5	5
9	0	1	1	2	3	3	3	3	4	4	5	5	5
10	0	1	1	2	3	4	4	4	4	5	5	5	6
11	0	1	1	2	3	4	5	5	5	5	5	5	6

```
def lcs(str1, str2):
    L,n,m = lcs_table(str1,str2)
    print_table(L,n,m)
    j = n
    k = m
    seq = [''] * L[n][m]
    while L[j][k] > 0:
        if L[j-1][k] < L[j][k] and L[j][k-1] < L[j][k]:
            j = j-1
            k = k-1
            seq[L[j][k]] = str1[j]
        elif L[j-1][k] == L[j][k]:
            j = j-1
        else:
            k = k-1
    print(L[n][m], ':', end=' ')
    for x in seq:
        print(x, end=' ')
```

This solution method is known as *Dynamic Programming*.

The goal should be to find some optimal answer.

There should be an easy way to break the problem down into simpler subproblems.

The optimal answer should be built easily from the optimal answers to the simpler versions of the problem.

We should be able to reuse the subproblem solutions more than once.

We have already seen another example of dynamic programming ...

Next lecture

Matching