2 Pairwise alignment

- Introduction to the pairwise sequence alignment problem
- Dot plots
- Scoring schemes
- The principle of dynamic programming
- Alignment algorithms based on dynamic programming

2.1 References

This exposition was developed by Clemens Gröpl. It is based on earlier versions with contributions by Daniel Huson, Knut Reinert, and Gunnar Klau.

- R. Durbin, S. Eddy, A. Krogh, G. Mitchison: *Biological sequence analysis*. Cambridge University Press, 1998. ISBN 0-521-62971-3 (Chapter 2)
- Neil C. Jones, Pavel A. Pevzner: *An Introduction to Bioinformatics Algorithms*. MIT Press, Cambridge, MA, 2004. ISBN 0-262-10106-8
- D. Gusfield: *Algorithms on Trees, Strings, and Sequences. Computer Science and Computational Biology*. Cambridge University Press, 1997, p.212ff
- Chao, Zhang: *Sequence comparison*. Chapter 2 and 3.

2.2 Importance of sequence alignment

The first fact of biological sequence analysis:
In biomolecular sequences (DNA, RNA, or amino acid sequences), high sequence similarity usually implies significant functional or structural similarity.

Duplication with modification:
The vast majority of extant proteins are the result of a continuous series of genetic duplications and subsequent modifications. As a result, redundancy is a built-in characteristic of protein sequences, and we should not be surprised that so many new sequences resemble already known sequences. . . . all of biology is based on enormous redundancy. . .

"We didn't know it at the time, but we found everything in life is so similar, that the same genes that work in flies are the ones that work in humans."
(Eric Wieschaus, cowinner of the 1995 Nobel prize in medicine for work on the genetics of *Drosophila* development.)

[Dan Gusfield, 1997, 212 ff]

Sequence alignment is the procedure of comparing two sequence (pairwise alignment) or more sequences (multiple alignment) by searching for a series of individual characters or character patterns that are in the same order in both sequences.

Two sequences are aligned by writing them across a page in two rows. Identical or similar characters are placed in the same column, whereas non-identical characters are either placed in the same column as a mismatch or are opposite a gap in the other sequence.

```
Two strings:          ⟶     Alignment:
IMISSMISSISSIPPI            I-MISSMISSISIPPI-
                              |||| ||   ||||
MYMISSISAHIPPIE             MYMISS-ISAH-IPPIE
```

Comparative Genomics

- Gene finding and function determination:
  Compare sequence to genes of known function. Two success stories:
  - Example 1: *v-sis-oncogene*[1] (discovered 1984). Comparison with all known genes led to striking similarity with regular growth gene. Conclusion: oncogene responsible for growth at the wrong time.
  - Example 2: *cystic fibrosis gene* (discovered 1989). Location narrowed down to 10^6 nucleotides on chromosome 7. Compared to all genes → similarity with gene for ATP binding proteins. Shed light on the nature of cystic fibrosis.

- High sequence similarity (e. g., human—mouse: 97%) between species allows to study other organisms in order to understand humans (e. g., Waardenburg's syndrome)
- Derive information about common origin (evolutionary trees)
- ...

2.3 Terminology

```
Two strings:          ⟶     Alignment:
IMISSMISSISSIPPI            I-MISSMISSISIPPI-
                              |||| ||   ||||
MYMISSISAHIPPIE             MYMISS-ISAH-IPPIE
```

I: Isoleucine, M: Methionine, S: Serine, P: Proline, Y: Tyrosine, A: Alanine, H: Histidine, E: Glutamic Acid, . . .

Two sequences are (globally) aligned by writing them across a page in two rows. Identical (or similar) characters are placed in the same column and called *matches*. Non-identical characters are either placed in the same column as a *mismatch*, or they are opposite to a *gap* in the other sequence. Also, we disallow columns that consist of gaps only.

One way to formalize pairwise sequence alignment is as follows: We are given two sequences $x = (x_1, x_2, \ldots, x_m)$ and $y = (y_1, y_2, \ldots, y_n)$ over an alphabet Σ. Let $\texttt{-} \notin \Sigma$ be the *gap symbol*, also called *space*. Let $h\colon (\Sigma \cup \{\texttt{-}\})^* \to \Sigma^*$ be the mapping that removes all gap symbols from a sequence over the alphabet $\Sigma \cup \{\texttt{-}\}$.

For example,

$$h\Big(\texttt{MYMISS-ISAH-IPPIE}\Big) = \texttt{MYMISSISAHIPPIE}.$$

[1]oncogenes are virus genes that cause cancer-like transformation of infected cells

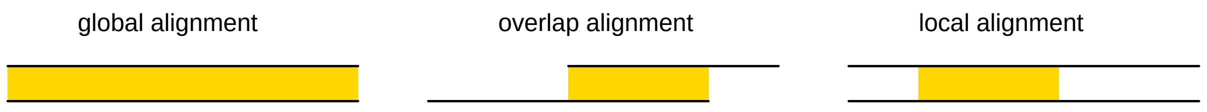
Then a *global alignment* of x and y is a pair of sequences x', y' such that

$$h(x') = x, \quad h(y') = y, \quad |x'| = |y'|, \quad \text{and} \quad (x'_i, y'_i) \neq (-,-) \text{ for all } i.$$

Here $|\cdot|$ denotes the length.

- *Match* - same (or similar) letter in both rows
- *Mismatch* - different letters in both rows
- *Insertion* - the letter opposite to a space
- *Deletion* - the space opposite to a letter
- *Indel* - a column containing a space

Depending on the input data, there are a number of different variants of alignment that are considered, among them *global alignment*, *overlap alignment*, and *local alignment*.



In an overlap alignment, we do not charge the end gaps (hence it is also called end-gap free alignment).

A local alignment is the same as a global alignment of two substrings of the sequences.

2.4 Finding alignments

How many alignments are there? Answer: many

How can we find a good pairwise alignment?

Why not simply *visualize* the data?

2.5 Dot plots

We can draw a matrix spanned up by two sequences and place a dot in each cell for which the correspondig symbols match. Stretches of matching symbols will show up as diagonal lines this way.

```
  IMISSMISSISSIPPI
M  *   *
Y
M  *   *
I * *   *  *  *  *
S    **  ** **
S    **  ** **
I * *   *  *  *  *
S    **  ** **
A
H
I * *   *  *  *  *
P              **
P              **
I * *   *  *  *  *
E
```

To obtain cleaner pictures, a *window size w* and a *stringency s* are used: A dot is only drawn at point (x, y) if within w positions around it there are at least s matches.

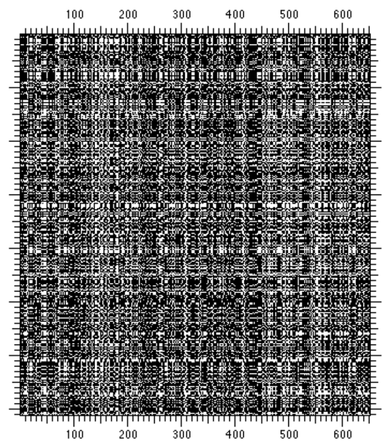
	I	M	I	S	S	M	I	S	S	I	S	I	P	P	I
M	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0
Y	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
M	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0
I	1	0	1	0	0	0	1	0	0	1	0	1	0	0	1
S	0	0	0	1	1	0	0	1	1	0	1	0	0	0	0
S	0	0	0	1	1	0	0	1	1	0	1	0	0	0	0
I	1	0	1	0	0	0	1	0	0	1	0	1	0	0	1
S	0	0	0	1	1	0	0	1	1	0	1	0	0	0	0
A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
H	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
I	1	0	1	0	0	0	1	0	0	1	0	1	0	0	1
P	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
P	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
I	1	0	1	0	0	0	1	0	0	1	0	1	0	0	1
E	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

$w = 1, s = 1$

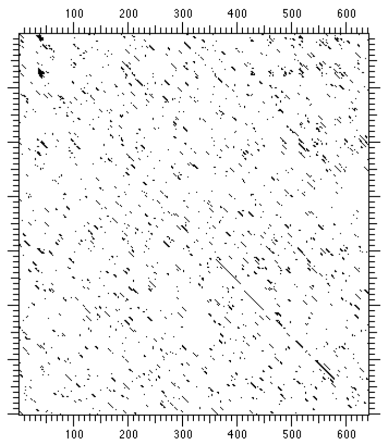
	I	M	I	S	S	M	I	S	S	I	S	I	P	P	I
M	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0
Y	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0
M	0	2	0	0	0	2	0	0	1	0	1	0	0	1	0
I	1	0	3	1	0	0	3	1	0	2	0	1	0	0	1
S	0	1	1	3	1	0	1	3	1	1	2	0	1	0	0
S	0	1	0	1	2	2	0	1	3	1	2	1	0	1	0
I	1	0	2	1	1	1	2	1	1	3	0	2	0	0	1
S	0	1	0	2	1	0	0	2	1	0	2	0	1	0	0
A	0	0	0	0	1	1	0	0	1	1	0	1	0	0	0
H	0	1	0	0	0	1	0	0	1	0	1	0	0	1	0
I	1	0	1	0	0	0	1	0	0	1	0	2	1	0	1
P	0	1	0	1	0	0	0	1	0	0	1	1	3	1	0
P	0	1	0	0	0	1	0	0	1	0	1	0	1	3	1
I	1	0	1	0	0	0	1	0	0	1	0	1	0	1	2
E	0	1	0	1	0	0	0	1	0	0	1	0	1	0	0

$w = 3, s = 3$

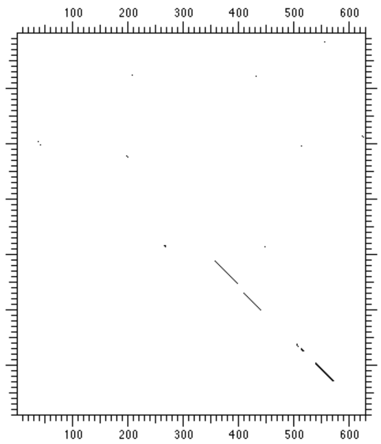
Here is a biological example (DNA sequences which encode the phage lambda and P22 repressor sequences):



$w = 1, s = 1$



$w = 11, s = 7$



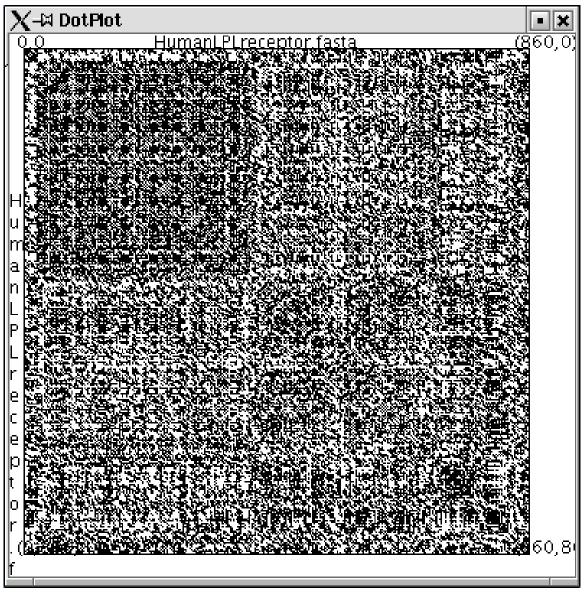
$w = 23, s = 15$

Tools on the web to play around with:

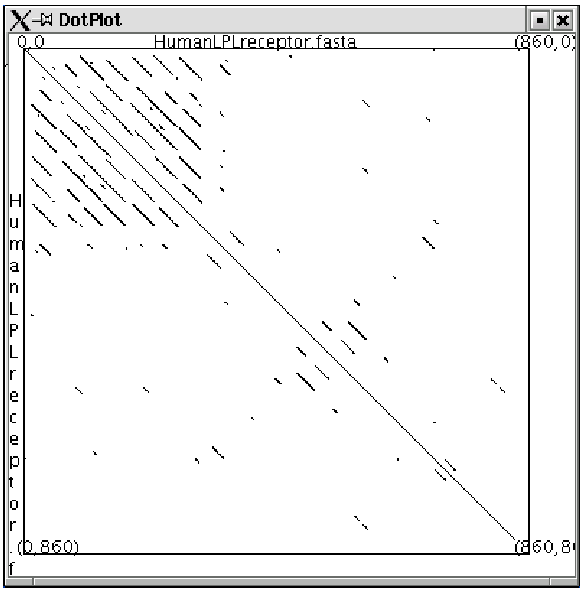
- `http://arbl.cvmbs.colostate.edu/molkit/dnadot/`

2.6 Repeat detection using dot plots

Here is another one. These dot plots of the human LDL receptor (protein sequence) against itself reveal many *repeats* in the first 300 positions.



$w = 1, s = 1$



$w = 23, s = 7$

2.7 How to score it?

So we *see* there is an alignment, but do we really *have* it, written in two rows?

To come up with an algorithm, we need a formal concept when an alignment is "good", i.e., *significant*. Let us have a look at some good and bad examples before we formally introduce a *scoring scheme*.

2.8 Significance of alignments

In the alignments below, the middle row contains a letter for identical amino acids, and a + if the amino acids are similar.

1. An alignment between very similar human alpha- and beta hemoglobins:

```
HBA_HUMAN   GSAQVKGHGKKVADALTNAVAHVDDMPNALSALSDLHAHKL
            G+ +VK+HGKKV  A+++++AH+D++ +++++LS+LH  KL
HBB_HUMAN   GNPKVKAHGKKVLGAFSDGLAHLDNLKGTFATLSELHCDKL
```

2. Plausible alignment to leghaemoglobin from yellow lupin:

```
HBA_HUMAN   GSAQVKGHGKKVADALTNAVAHV---D--DMPNALSALSDLHAHKL
            ++ ++++H+ KV   + +A  ++          +L+ L+++H+ K
LGB2_LUPLU  NNPELQAHAGKVFKLVYEAAIQLQVTGVVVTDATLKNLGSVHVSKG
```

3. A spurious high-scoring alignment of human alpha globin to a nematode glutathione *S*-transferase homologue:

```
HBA_HUMAN   GSAQVKGHGKKVADALTNAVAHVDDMPNALSALSD----LHAHKL
            GS+ + G +   +D L  ++ H+ D+  A +AL D    ++AH+
F11G11.2    GSGYLVGDSLTFVDLL--VAQHTADLLAANAALLDEFPQFKAHQE
```

In (1), there are many positions at which the two corresponding residues are identical. Many others are functionally conservative. E.g., the `D-E` pair towards the end: both negatively charged amino acids.

In (2), we also see a biologically meaningful alignment, as it is known that the two proteins are evolutionarily related, have the same 3D structure and both have the same function. However, there are many fewer identities and gaps have been introduced in the sequences.

In (3), we see an alignment with a similar number of identities or conservative changes. However, this is a spurious alignment between two proteins that have completely different structure and function.

2.9 Scoring schemes

The basic mutational processes are *substitutions*, *insertions*, and *deletions*. Substitutions give rise to *mismatches*. Insertions and deletions give rise to *gaps*.

The *total score* assigned to an alignment is the sum of

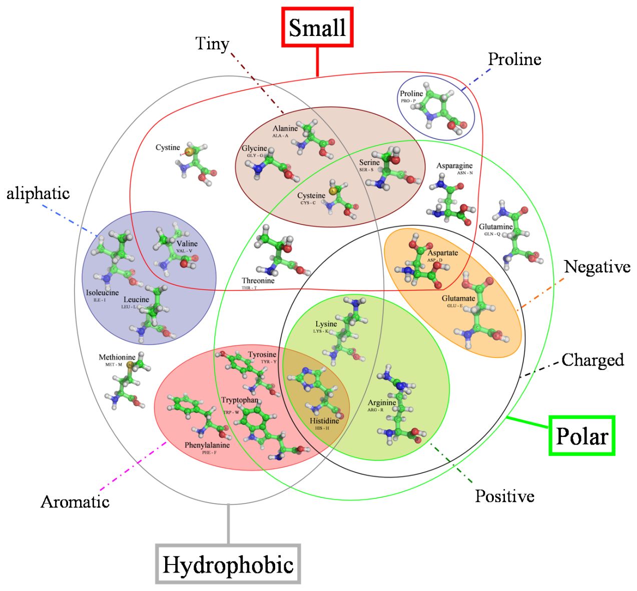
1. the sum of terms for matches and mismatches,
2. the sum of terms for gaps (i.e., indels).

Formally we can treat the space character just like any other. Then we obtain the following additive scoring scheme:

The score of an alignment (x', y') is
$$\sum_i \delta(x'_i, y'_i),$$
where $\delta\colon (\Sigma \cup \{-\})^2 \to \mathbb{R}$ is a *score matrix*.

2.10 Classification of amino acids

How should we choose the score matrix δ? Amino acids can be grouped according to chemical properties.



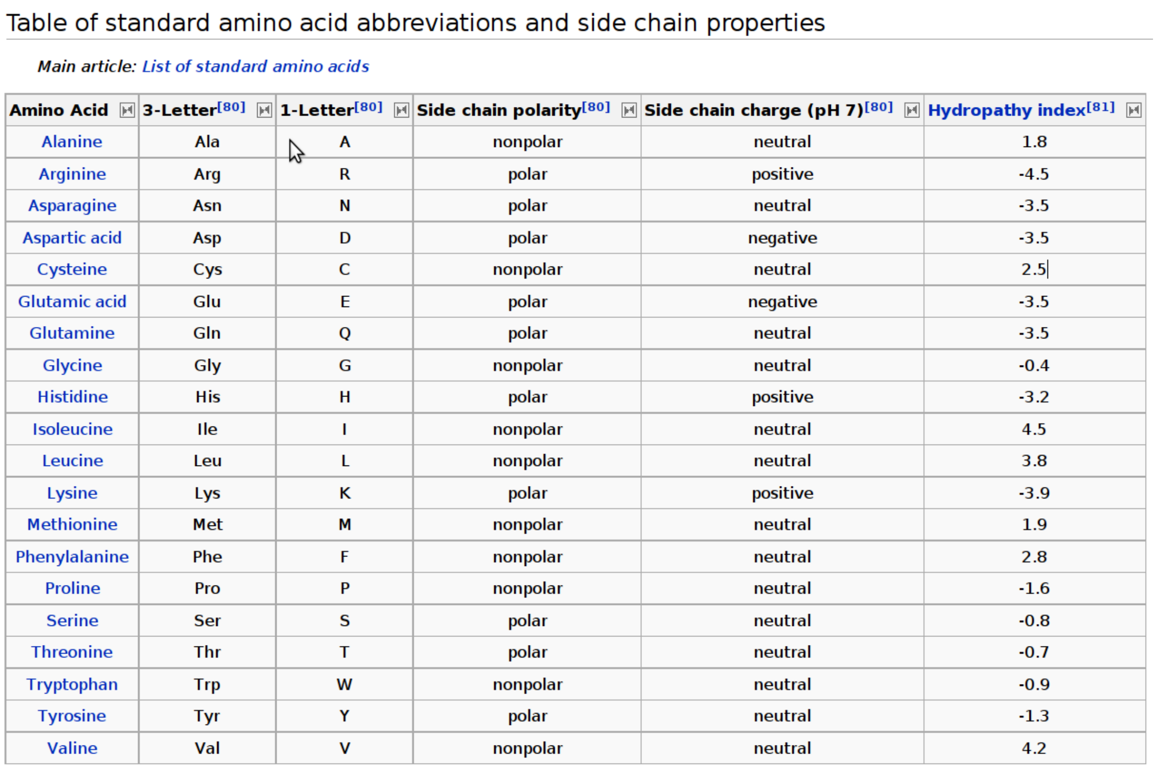


Table of standard amino acid abbreviations and side chain properties

Main article: List of standard amino acids

Amino Acid	3-Letter[80]	1-Letter[80]	Side chain polarity[80]	Side chain charge (pH 7)[80]	Hydropathy index[81]
Alanine	Ala	A	nonpolar	neutral	1.8
Arginine	Arg	R	polar	positive	-4.5
Asparagine	Asn	N	polar	neutral	-3.5
Aspartic acid	Asp	D	polar	negative	-3.5
Cysteine	Cys	C	nonpolar	neutral	2.5
Glutamic acid	Glu	E	polar	negative	-3.5
Glutamine	Gln	Q	polar	neutral	-3.5
Glycine	Gly	G	nonpolar	neutral	-0.4
Histidine	His	H	polar	positive	-3.2
Isoleucine	Ile	I	nonpolar	neutral	4.5
Leucine	Leu	L	nonpolar	neutral	3.8
Lysine	Lys	K	polar	positive	-3.9
Methionine	Met	M	nonpolar	neutral	1.9
Phenylalanine	Phe	F	nonpolar	neutral	2.8
Proline	Pro	P	nonpolar	neutral	-1.6
Serine	Ser	S	polar	neutral	-0.8
Threonine	Thr	T	polar	neutral	-0.7
Tryptophan	Trp	W	nonpolar	neutral	-0.9
Tyrosine	Tyr	Y	polar	neutral	-1.3
Valine	Val	V	nonpolar	neutral	4.2

The BLOSUM50 matrix:

```
   A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V  B  Z  X  *
A  5 -2 -1 -2 -1 -1 -1  0 -2 -1 -2 -1 -1 -3 -1  1  0 -3 -2  0 -2 -1 -1 -5
R -2  7 -1 -2 -4  1  0 -3  0 -4 -3  3 -2 -3 -3 -1 -1 -3 -1 -3 -1  0 -1 -5
N -1 -1  7  2 -2  0  0  0  1 -3 -4  0 -2 -4 -2  1  0 -4 -2 -3  4  0 -1 -5
D -2 -2  2  8 -4  0  2 -1 -1 -4 -4 -1 -4 -5 -1  0 -1 -5 -3 -4  5  1 -1 -5
C -1 -4 -2 -4 13 -3 -3 -3 -3 -2 -2 -3 -2 -2 -4 -1 -1 -5 -3 -1 -3 -3 -2 -5
Q -1  1  0  0 -3  7  2 -2  1 -3 -2  2  0 -4 -1  0 -1 -1 -1 -3  0  4 -1 -5
E -1  0  0  2 -3  2  6 -3  0 -4 -3  1 -2 -3 -1 -1 -1 -3 -2 -3  1  5 -1 -5
G  0 -3  0 -1 -3 -2 -3  8 -2 -4 -4 -2 -3 -4 -2  0 -2 -3 -3 -4 -1 -2 -2 -5
H -2  0  1 -1 -3  1  0 -2 10 -4 -3  0 -1 -1 -2 -1 -2 -3  2 -4  0  0 -1 -5
I -1 -4 -3 -4 -2 -3 -4 -4 -4  5  2 -3  2  0 -3 -3 -1 -3 -1  4 -4 -3 -1 -5
L -2 -3 -4 -4 -2 -2 -3 -4 -3  2  5 -3  3  1 -4 -3 -1 -2 -1  1 -4 -3 -1 -5
K -1  3  0 -1 -3  2  1 -2  0 -3 -3  6 -2 -4 -1  0 -1 -3 -2 -3  0  1 -1 -5
M -1 -2 -2 -4 -2  0 -2 -3 -1  2  3 -2  7  0 -3 -2 -1 -1  0  1 -3 -1 -1 -5
F -3 -3 -4 -5 -2 -4 -3 -4 -1  0  1 -4  0  8 -4 -3 -2  1  4 -1 -4 -4 -2 -5
P -1 -3 -2 -1 -4 -1 -1 -2 -2 -3 -4 -1 -3 -4 10 -1 -1 -4 -3 -3 -2 -1 -2 -5
S  1 -1  1  0 -1  0 -1  0 -1 -3 -3  0 -2 -3 -1  5  2 -4 -2 -2  0  0 -1 -5
T  0 -1  0 -1 -1 -1 -1 -2 -2 -1 -1 -1 -1 -2 -1  2  5 -3 -2  0  0 -1  0 -5
W -3 -3 -4 -5 -5 -1 -3 -3 -3 -3 -2 -3 -1  1 -4 -4 -3 15  2 -3 -5 -2 -3 -5
Y -2 -1 -2 -3 -3 -1 -2 -3  2 -1 -1 -2  0  4 -3 -2 -2  2  8 -1 -3 -2 -1 -5
V  0 -3 -3 -4 -1 -3 -3 -4 -4  4  1 -3  1 -1 -3 -2  0 -3 -1  5 -4 -3 -1 -5
B -2 -1  4  5 -3  0  1 -1  0 -4 -4  0 -3 -4 -2  0  0 -5 -3 -4  5  2 -1 -5
Z -1  0  0  1 -3  4  5 -2  0 -3 -3  1 -1 -4 -1  0 -1 -2 -2 -3  2  5 -1 -5
X -1 -1 -1 -1 -2 -1 -1 -2 -1 -1 -1 -1 -1 -2 -2 -1  0 -3 -1 -1 -1 -1 -1 -5
* -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5

The "extra" four columns are
 B wildcard for N or D
 Z wildcard for Q or E
 X wildcard for any amino acid
 * stop codon (not usually used much in protein-protein scoring
 but handy for scoring translated sequences)
```

Later we will explain the derivation of the scoring schemes in more detail. For now, let us assume that each substitition, insertion and deletion has a specific score. If we use the Blosum50 matrix, we can compute the score of the following alignment:

```
HBA_HUMAN  G  S  A  Q  V  K  G  H  G  K  K  V
           G  +     +  V  K  +  H  G  K  K  V
HBB_HUMAN  G  N  P  K  V  K  A  H  G  K  K  V
score      8 +1 -1 +2 +5 +6 +0+10 +8 +6 +6 +5 = 56
```

2.11 Gap penalties

Hamming distance and Levenshtein distance

In the simplest case, we do not allow any gaps at all and charge all mismatches at unit cost. This is called the *Hamming distance.* The alignment is forced to be on one diagonal of the dot plot.

If we charge all insertions, deletions, and mismatches at unit cost, we obtain the *Levenshtein distance.* This measure is also called the *edit distance,* because it counts the number of elementary edit operations needed to transform one sequence into the other.

Both scoring schemes have applications in biological sequence analysis, especially for nucleic acids. Often, however, it is better to

1. distinguish the type of a mismatch, and
2. take the length of consecutive gaps into account.

Linear and affine gap cost

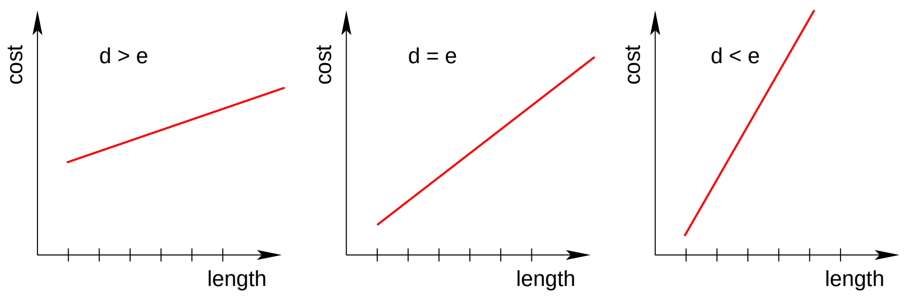
Gaps in an alignment are undesirable and thus penalized. In its simplest form, the cost associated with a gap of length $g \geq 1$ is given by a *linear* score,

$$\gamma(g) = -g \cdot d\,.$$

An *affine* score, however,

$$\gamma(g) = -d - (g-1)e$$

often produces better results. Here d is the *gap open* penalty and e is the *gap extension* penalty.



Usually one sets $e < d$, i.e., there is a large penalty for opening a gap, but a smaller penalty for extending it. Then affine gap costs favor alignments with fewer but larger gaps. For example:

Using linear gap penalties:

```
GSAQVKGHGKKVADALTNAVAHVDDMPNALSALSDLHAHKL
GSAQVKGHGKK--------VA--D----A-SALSDLHAHKL
```

Using affine gap penalties:

```
GSAQVKGHGKKVADALTNAVAHVDDMPNALSALSDLHAHKL
GSAQVKGHGKKVADA---------------SALSDLHAHKL
```

The case $d < e$ is sometimes used when comparing output of DNA sequencing machines. There it happens frequently that single bases are left out near the end of a read.

2.12 Remarks on scoring schemes

1. The scoring schemes we have seen so far are based solely on primary structure. This is a reasonable approximation especially for DNA. For RNA, one observes that bases which are coupled by secondary structure are highly correlated. But even for proteins the primary structure can tell us a lot.
2. We will not explain the probabilistic background of the additive scoring scheme (in this lecture), but simply take it as granted. Thus we will not explain, e.g., how the BLOSUM50 matrix was derived from experimental data, assuming a certain model of evolution, etc.

2.13 Alignment algorithms

Given a scoring scheme and two input sequences, we need an algorithm to compute the highest-scoring alignment of the sequences.

We will discuss alignment algorithms based on *dynamic programming*. Dynamic programming algorithms play a central role in computational sequence analysis. They are guaranteed to find the highest-scoring alignment.

Note: For large sequences exact "DP algorithms" can be too slow and *heuristics* (such as BLAST, FASTA, MUMMER, QUASAR,...) are then used which perform very well in most cases, but will miss the best alignment for some sequence pairs.

Paradigm: Dynamic Programming

Dynamic programming is a powerful algorithmic design principle.

Principle
Compute solutions of "bigger" problems by combining solutions of smaller subproblems. Storing all solutions avoids recomputing the same quantity over and over again, and a potential exponential blow-up in the running time.

Well-suited if subproblems share subsubproblems.

Typically applied to *optimization problems*. Three components:

1. Recursively define the value of an optimal solution
2. Compute values of subproblems in bottom-up fashion (storing the solution values)
3. Construct solution (traceback)

Examples
Fibonacci numbers[2], knapsack, longest common subsequence, alignments in database search (BLAST, FASTA), multiple alignments (clustalW), gene finding (GENSCAN), RNA-folding (mfold), phylogenetic inference (PHYLIP), . . .

The Fibonacci code discussed in the exercises is available from the web page:
`http://www.alice-dsl.net/clemens.groepl/restricted/vl_genomanalyse/files.html`

```
long int fibo_full_recursion( long int n )
{
  if ( n == 0 ) return 0;
  if ( n == 1 ) return 1;
  return fibo_full_recursion(n-1) + fibo_full_recursion(n-2);
}

long int fibo_dyn_prog( long int n )
{
  if ( n == 0 ) return 0;
  if ( n == 1 ) return 1;
  std::vector<long int> fibo_vec;
  fibo_vec.resize(n+1);
  fibo_vec[0] = 0;
  fibo_vec[1] = 1;
  for ( int i = 2; i <= n; ++i )
  {
    fibo_vec[i] = fibo_vec[i-1] + fibo_vec[i-2];
  }
  return fibo_vec[n];
}
```

2.14 Global alignment: Needleman-Wunsch algorithm

(Saul Needleman and Christian Wunsch, 1970; improved by Peter Sellers, 1974)

[2]blackboard

Consider the problem of finding the optimal *global alignment* of two sequences x and y. The Needleman-Wunsch algorithm is a "dynamic program" that solves this problem. What does this mean?

The **idea** is to build up a table of optimal alignments for all pairs of prefixes of x and y using (already known) optimal alignments of shorter prefixes of x and y.

We are given two sequences $x = (x_1, x_2, \dots, x_m)$ and $y = (y_1, y_2, \dots, y_n)$. We will compute a matrix, usually called *dynamic programming table*,

$$F : \{0, 1, 2, \dots, m\} \times \{0, 1, 2, \dots, n\} \to \mathbb{R}$$

in which $F(i, j)$ equals the best score of an alignment of the two prefixes $(x_1, x_2, \dots, x_i)$ and $(y_1, y_2, \dots, y_j)$.

Outline of the algorithm: We fill the table F recursively, bottom-up[3]. We start with initial cases like $F(0, 0) = 0$ and then compute each $F(i, j)$ from $F(i - 1, j - 1)$, $F(i - 1, j)$ and $F(i, j - 1)$:

$$\begin{array}{c|cccccccc}
 & & x_1 & x_2 & \dots & x_{i-1} & & x_i & \dots \; x_m \\
\hline
 & F(0,0) & F(1,0) & F(2,0) & & & & \vdots & \\
y_1 & F(0,1) & & & & & & \vdots & \\
y_2 & F(0,2) & & & & & & \vdots & \\
\vdots & & & & & & & \vdots & \\
y_{j-1} & & & & & F(i-1,j-1) & & F(i,j-1) & \\
 & & & & & & \nwarrow & \uparrow & \\
y_j & \dots & \dots & \dots & \dots & F(i-1,j) & \leftarrow & \boxed{F(i,j)} & \\
\vdots & & & & & & & & \\
y_n & & & & & & & &
\end{array}$$

This is applied until the whole matrix F is filled with values. Then $F(m, n)$ is the score of the best global alignment.

Why *can* we apply such a **recursion**?

There are three ways how the last column of an alignment of $(x_1, x_2, \dots, x_i)$ and $(y_1, y_2, \dots, y_j)$ can look like:

x_i aligns to y_j:	x_i aligns to a gap:	y_j aligns to a gap:
$\begin{array}{cccc} \mathtt{I} & \mathtt{G} & \mathtt{A} & \boxed{x_i} \\ \mathtt{L} & \mathtt{G} & \mathtt{V} & \boxed{y_j} \end{array}$	$\begin{array}{ccccc} \mathtt{A} & \mathtt{I} & \mathtt{G} & \mathtt{A} & \boxed{x_i} \\ \mathtt{G} & \mathtt{V} & \mathtt{C} & y_j & \boxed{-} \end{array}$	$\begin{array}{ccccc} \mathtt{G} & \mathtt{A} & x_i & - & \boxed{-} \\ \mathtt{S} & \mathtt{L} & \mathtt{G} & \mathtt{V} & \boxed{y_j} \end{array}$

We obtain $F(i, j)$ as the largest score that arises from one of these cases[4]:

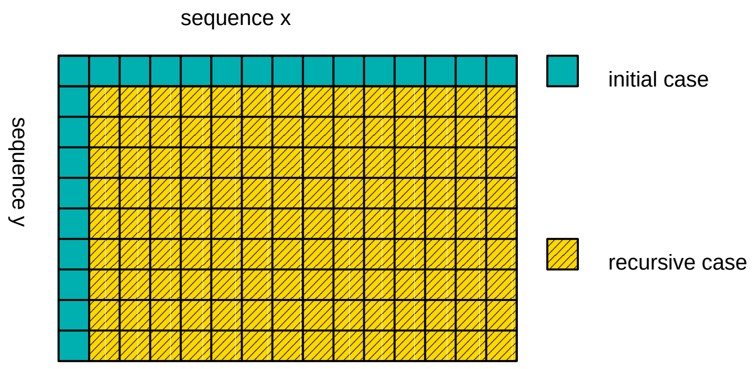
$$F(i, j) := \max \begin{cases} F(i - 1, j - 1) + s(x_i, y_j) \\ F(i - 1, j) - d \\ F(i, j - 1) - d \end{cases}$$

To complete the description of the recursion, we need to set the **initial values** on the upper and the left boundary, $F(i, 0)$ and $F(0, j)$:

We set $F(i, 0) = i \cdot d$ for $i = 0, 1, \dots, m$ and $F(0, j) = j \cdot d$ for $j = 0, 1, \dots, n$.

[3] This corresponds to step 2 on the DP paradigm slides

[4] This corresponds to step 1 on the DP paradigm slides



The final value $F(m,n)$ is the *score* of the best global alignment between x and y.

To obtain an *alignment* corresponding to this score, we still need to find the path of choices that has led the recursion to the final score. This is called a *traceback*[5]

However this is actually easy, as we only have to store one of the symbols

$$T(i,j) \in \{\leftarrow, \nwarrow, \uparrow\}$$

(or a subset thereof) whenever we assign a value to a "DP entry" $F(i,j)$.

2.15 Needleman-Wunsch algorithm

Input: two sequences x and y
Output: optimal alignment and score α

```
Initialization:
Set $F(0,0) := 0$
Set $F(i,0) := -i \cdot d$ and $T(i,0) := (i-1,0)$ for all $i = 1,2,\dots,m$
Set $F(0,j) := -j \cdot d$ and $T(0,j) := (0,j-1)$ for all $j = 1,2,\dots,n$
Recurrence:
for $i = 1,2,\dots,m$ do:
    for $j = 1,2,\dots,n$ do:
        Set $F(i,j) := \max \begin{cases} F(i-1,j-1) + s(x_i,y_j) \\ F(i-1,j) - d \\ F(i,j-1) - d \end{cases}$
        Set backtrace $T(i,j)$ to the maximizing pair $(i',j')$ (this can be encoded e.g. by $\{\nwarrow, \leftarrow, \uparrow\}$)
The best score is $\alpha := F(m,n)$
Traceback:
Set $(i,j) := (m,n)$
repeat
    if $T(i,j) = (i-1,j-1)$ print $\binom{x_i}{y_j}$
    else if $T(i,j) = (i-1,j)$ print $\binom{x_i}{-}$ else print $\binom{-}{y_j}$
    Set $(i,j) := T(i,j)$
until $(i,j) = (0,0)$
```

Note that alternative strategies are possible for implementing the traceback.

- As shown in the pseudocode above, we can store the indices of the preceding matrix cell: $T(i,j) \in \{(i-1,j-1),(i,j-1),(i-1,j)\}$. As a slight variation of the same idea, we can encode the traceback direction

[5]This corresponds to step 3 on the DP paradigm slides.

using symbols: $T(i, j) \in \{\leftarrow, \nwarrow, \uparrow\}$. Note that the matrix T occupies $O(mn)$ memory, but only $O(m + n)$ of it is actually used during traceback.

- We can eliminate the additional array T completely and distinguish among the three cases by other means. Namely, we redo the calculation of $F(i, j) = \max\{\ldots\}$ in order to find out the direction for traceback. This way we have a larger (but still constant) amount of work to perform for each column of the alignment, which adds up to $O(m + n)$ for the entire alignment, but we avoid $O(mn)$ work for maintaining T.

2.16 An example of global alignment

We will use two short amino acid sequences for illustration:

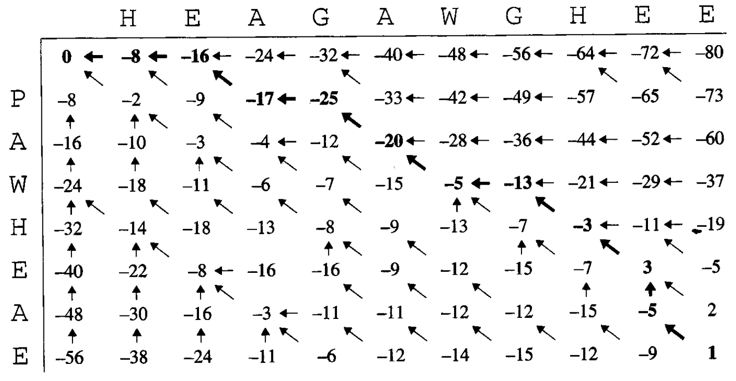
`HEAGAWGHEE` and `PAWHEAE`.

To score the alignment we will use the BLOSUM50 matrix and a gap cost of $d = 8$.

BLOSUM50 values:

	H	E	A	G	A	W	G	H	E	E
P	-2	-1	-1	-2	-1	-4	-2	-2	-1	-1
A	-2	-1	**5**	0	**5**	-3	0	-2	-1	-1
W	-3	-3	-3	-3	-3	**15**	-3	-3	-3	-3
H	**10**	0	-2	-2	-2	-3	-3	**10**	0	0
E	0	**6**	-1	-3	-1	-3	-3	0	**6**	**6**
A	-2	-1	**5**	0	**5**	-3	0	-2	-1	-1
E	0	**6**	-1	-3	-1	-3	-3	0	**6**	**6**

DP matrix:



		H	E	A	G	A	W	G	H	E	E
	0	**–8**	**–16**	–24	–32	–40	–48	–56	–64	–72	–80
P	–8	–2	–9	**–17**	**–25**	–33	–42	–49	–57	–65	–73
A	–16	–10	–3	–4	–12	**–20**	–28	–36	–44	–52	–60
W	–24	–18	–11	–6	–7	–15	**–5**	**–13**	–21	–29	–37
H	–32	–14	–18	–13	–8	–9	–13	–7	**–3**	–11	–19
E	–40	–22	–8	–16	–16	–9	–12	–15	–7	**3**	–5
A	–48	–30	–16	–3	–11	–11	–12	–12	–15	**–5**	2
E	–56	–38	–24	–11	–6	–12	–14	–15	–12	–9	**1**

```
HEAGAWGHE-E
--P-AW-HEAE
```

Durbin *et al.* (1998)

2.17 Web ressources

There is a nice Java applet illustrating the NW algorithm on the web:
`http://lectures.molgen.mpg.de/PracticalSection/AliApplet/index.html`

EMBOSS is a comprehensive suite of bioinformatics tools.
`http://emboss.sourceforge.net/index.html`

Many organizations provide online interfaces to the EMBOSS tools, e.g.
`http://emboss.bioinformatics.nl/`

2.18 Complexity of the Needleman-Wunsch algorithm

We need to store $(n+1)(m+1)$ numbers. Each number takes a constant number of calculations to compute: just three sums and a max.

Hence, the algorithm requires $O(nm)$ time and memory.

2.19 Arbitrary gap costs

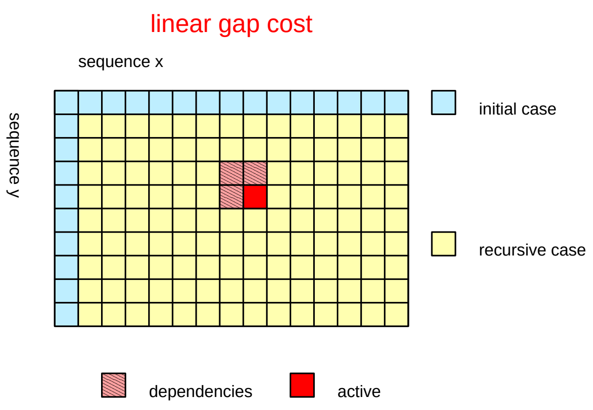
A way to deal with arbitrary gap costs is as follows. Assume a gap of length g has cost $\gamma(g)$. Then we can replace

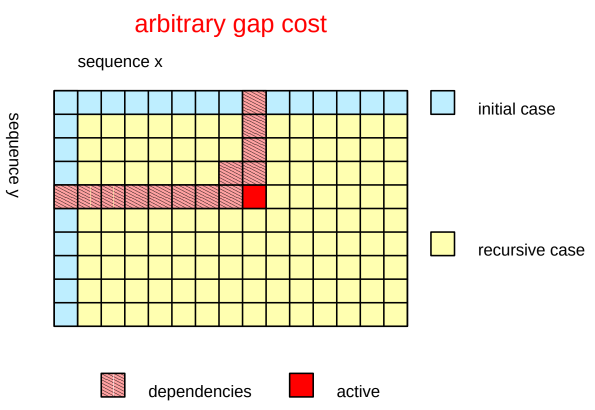
$$F(i,j) := \max \begin{cases} F(i-1,j-1) + s(x_i, y_j) \\ F(i-1,j) - d \\ F(i,j-1) - d \end{cases}$$

with

$$F(i,j) := \max \begin{cases} F(i-1,j-1) + s(x_i, y_j) & \\ F(i-g,j) - \gamma(g) & g = 1,\dots,i \\ F(i,j-g) - \gamma(g) & g = 1,\dots,j \end{cases}$$

However this increases the running time from $O(mn)$ to $O(mn \max\{m,n\})$, since we have much more **dependencies in the DP recurrence**:

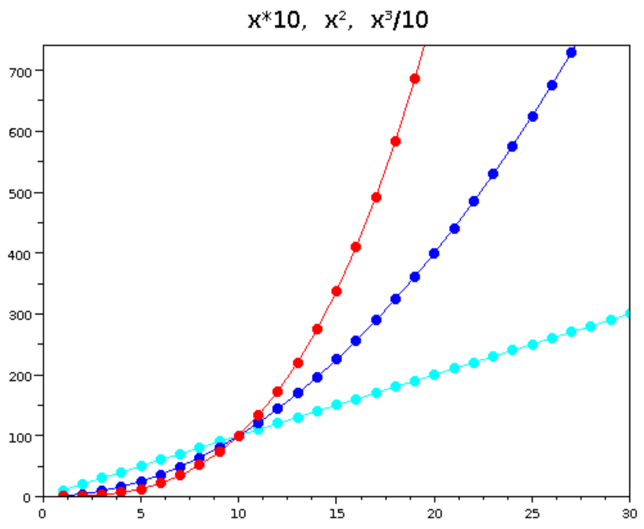




Remark: For *affine gap* costs there is are clever ways to do this in $O(mn)$.

2.20 Growth rates

For biological sequence analysis, we prefer algorithms that have time and space requirements that are linear in the length of the sequences. Quadratic time ($O(n^2)$) algorithms are a little slow, but feasible. Cubic time ($O(n^3)$) algorithms are feasible only for very short sequences.



2.21 Affine gap costs

The standard alternative to using the above recursion is to use an *affine gap score*

$$\gamma(g) = -d - (g-1)e,$$

with d the *gap-open score* and e the *gap-extension* score.

We will discuss how to modify the Needleman-Wunsch algorithm for global alignment so as to incorporate affine gap costs. The resulting algortithm is due to Osamu Gotoh (1982).

In the justification of the Needleman-Wunsch algorithm we made a case distinction based on the *last* column of an optimal alignment of the prefixes of both sequences. For affine gap costs, we will need to consider the *second last* column as well.

As a consequence, instead of using just one matrix $F(i, j)$ to represent the best score attainable up to x_i and y_j, we will now use three matrices M, I_x and I_y, and distinguish the state of the second last column.

Reminder (from Needleman-Wunsch): there are three ways how the last column of an alignment of $(x_1, x_2, \ldots, x_i)$ and $(y_1, y_2, \ldots, y_j)$ can look like:

x_i aligns to y_j:	x_i aligns to a gap:	y_j aligns to a gap:
I G A x_i	A I G A x_i	G A x_i - -
L G V y_j	G V C y_j -	S L G V y_j

We introduce three matrices:

1. $M(i, j)$ is the best score up to (i, j), given that x_i is aligned to y_j,
2. $I_x(i, j)$ is the best score up to (i, j), given that x_i is aligned to a gap, and
3. $I_y(i, j)$ is the best score up to (i, j), given that y_j is aligned to a gap.

Now we will distinguish the state of the second last column as well:

	x_i aligns to y_j:	x_i aligns to a gap:	y_j aligns to a gap:
M	I G A x_i	A I G A x_i	G A G x_i -
	L G V y_j	G V C y_j -	S L G V y_j
I_x	I G A x_i	A I G A x_i	G A G x_i -
	L G - y_j	G V y_j - -	S L G - y_j
I_y	I G - x_i	A I G - x_i	G x_i - - -
	L G V y_j	G V C y_j -	S L G V y_j

2.22 Gotoh algorithm

The cases in the gray boxes are undesirable because a gap in one sequence is immediately followed by a gap in the other. We will explicitly exclude them from consideration. (The optimal alignment does not use them anyway, if $-d - e$ is less than the lowest mismatch score. However the scoring scheme does not always have this property, so we are really enforcing a new requirement.)

From the remaining seven cases we obtain the following recursions:

Recursion:

$$M(i,j) = \max \begin{cases} M(i-1,j-1) + s(x_i,y_j), \\ I_x(i-1,j-1) + s(x_i,y_j), \\ I_y(i-1,j-1) + s(x_i,y_j); \end{cases}$$

$$I_x(i,j) = \max \begin{cases} M(i-1,j) - d, \\ I_x(i-1,j) - e; \end{cases}$$

$$I_y(i,j) = \max \begin{cases} M(i,j-1) - d, \\ I_y(i,j-1) - e. \end{cases}$$

The formulas for initialization at the upper and left margin ($i = 0$ respectively $j = 0$) are derived from the recursion. However, there are some "impossible cases", represented by $I_x(0,j)$ and $I_y(i,0)$. We assign a value of $-\infty$ to these matrix entries, such that they will not have an influence on the maximum computations.

Initialization:

$$M(0,0) = 0, \quad I_x(0,0) = I_y(0,0) = -\infty$$
$$I_x(i,0) = -d - (i-1)e, \quad M(i,0) = I_y(i,0) = -\infty, \text{ for } i = 1,\ldots,n$$
$$I_y(0,j) = -d - (j-1)e, \quad M(0,j) = I_x(0,j) = -\infty, \text{ for } j = 1,\ldots,m.$$

The traceback uses the same ideas as the Needleman-Wunsch algorithm. There are just a few more cases to consider. . . ;-)

2.23 Local alignment: Motivation

- Two genes in different species may be similar over short conserved regions and dissimilar over the remaining regions.
  Example:
  - *Homeobox genes* (which regulate embryonic development) have a short region called the *homeodomain* that is highly conserved between species.
  - A global alignment would not find the homeodomain because it would try to align the *entire* sequence
- Often we want to find the position of a fragment of DNA in a genomic sequence (with errors).

- DNA toy example:
  - *Global Alignment*:

```
--T--CC-C-AGT--TATGT-CAGGGGACACG-A-GCATGCAGA-GAC
  |  || |  ||  | | | |||    || | | |  | ||||   |
AATTGCCGCC-GTCGT-T-TTCAG----CA-GTTATG-T-CAGAT--C
```

  - *Local Alignment*—better suited to find conserved segment:

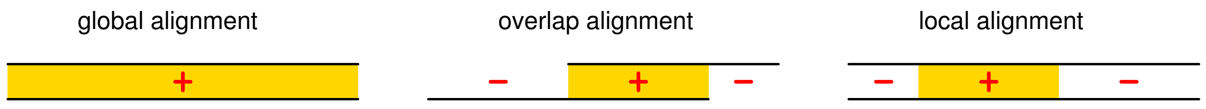
```
                  tccCAGTTATGTCAGgggacacgagcatgcagagac
                     ||||||||||||
aattgccgccgtcgttttcagCAGTTATGTCAGatc
```

2.24 Local alignment: Smith-Waterman algorithm

(Temple Smith and Mike Waterman, 1981)

A *local alignment* of two sequences x and y is a global alignment of a substring x' of x and a substring y' of y.

Technically speaking, why should we exclude some parts at the beginning and at the end of both sequences in a local alignment? If these parts of the sequences are evolutionary unrelated, even the best global alignment of $(x_1, \dots, x_i)$ and $(y_1, \dots, y_j)$ (and likewise, for the suffixes) will have a very bad score. As we have seen in the DNA toy example, this may even enforce that the optimal global alignment does not align the evolutionary related parts. Therefore we allow to omit these parts in the optimization of the score.



We modify the recurrence formula for $F(i, j)$ such that we can *start a local alignment at any place in the DP matrix*. This means, we replace

$$F(i,j) := \max \begin{cases} F(i-1, j-1) + s(x_i, y_j) \\ F(i-1, j) - d \\ F(i, j-1) - d \end{cases}$$

with

$$F(i,j) := \max \begin{cases} 0 \\ F(i-1, j-1) + s(x_i, y_j) \\ F(i-1, j) - d \\ F(i, j-1) - d \end{cases}$$

In the Smith-Waterman algorithm, any pair of indices (i, j) can represent the start of a local alignment. In the Needleman-Wunsch algorithms, this role that was reserved for the upper left corner $(0, 0)$.

Likewise, we *start the* backtrace *at a position* (k, ℓ) *that maximizes* $F(k, \ell)$, not necessarily at (m, n). This is written as

$$(k, \ell) := \arg\max\{F(k, \ell) \mid k = 0, \dots, m,\ \ell = 0, \dots, n\}$$

This way, the local alignment can stop before it reaches the end of the sequences.

The backtrace stops when we reach a position (i, j) such that $F(i, j) = 0$.

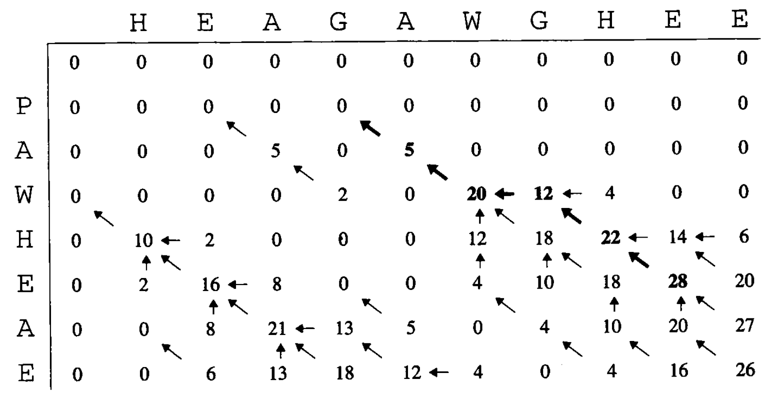
2.25 Smith-Waterman algorithm

Input: two sequences x and y
Output: optimal local alignment and score α

```
Initialization:
Set $F(0,0) := 0$ and $T(0,0) = \perp$ (where $\perp$ means "undefined")
Set $F(i,0) := 0$ and $T(i,0) := \perp$ for all $i = 1, 2, \dots, m$
Set $F(0,j) := 0$ and $T(0,j) := \perp$ for all $j = 1, 2, \dots, n$
Recurrence:
for $i = 1, 2, \dots, m$ do:
    for $j = 1, 2, \dots, n$ do:
        Set $F(i,j) := \max \begin{cases} 0 \\ F(i-1, j-1) + s(x_i, y_j) \\ F(i-1, j) - d \\ F(i, j-1) - d \end{cases}$
        Set backtrace $T(i,j)$ to the maximizing pair $(i', j')$, or "undefined" in the first case
        (encoded as $\in \{\perp, \nwarrow, \leftarrow, \uparrow\}$, respectively)
Set $(k, \ell) := \arg\max\{F(k, \ell) \mid k = 0, \dots, m,\ \ell = 0, \dots, n\}$
The best score is $\alpha := F(k, \ell)$
Traceback:
Set $(i, j) := (k, \ell)$
repeat
    if $T(i,j) = (i-1, j-1)$ print $\binom{x_i}{y_j}$
    else if $T(i,j) = (i-1, j)$ print $\binom{x_i}{-}$ else print $\binom{-}{y_j}$
    Set $(i, j) := T(i, j)$
until $F(i, j) = 0$
```

2.26 An example of local alignment

A local alignment matrix using $d = 8$ and BLOSUM50 scores:


H E A G A W G H E E
0 0 0 0 0 0 0 0 0 0 0
P 0 0 0 0 0 0 0 0 0 0 0
A 0 0 0 5 0 5 0 0 0 0 0
W 0 0 0 0 2 0 20 12 4 0 0
H 0 10 2 0 0 0 12 18 22 14 6
E 0 2 16 8 0 0 4 10 18 28 20
A 0 0 8 21 13 5 0 4 10 20 27
E 0 0 6 13 18 12 4 0 4 16 26

```
AWGHE
AW-HE
```

Durbin *et al.* (1998)

2.27 Scoring schemes, revisited

When comparing two biological sequences, we usually want to determine whether and how they diverged from a common ancestor by a process of mutation and selection.

The basic mutational processes are *substitutions*, *insertions* and *deletions*. The latter two give rise to *gaps*.

The total score assigned to an alignment is the sum of terms for each aligned pair of residues, plus terms for each gap.

In a probabilistic interpretation, this will correspond to the the logarithm of the relative likelihood that the sequences are related, compared to being unrelated.

Using such an additive scoring scheme is based on the assumption that mutations at different sites occur independently of each other. This is often reasonable for DNA and proteins, but not for structural RNA, where base pairing introduces very important long-range dependences.

2.28 Substitution matrices

First we consider score terms for aligned residue pairs.

Assume we are given two sequences $x = (x_1, x_2, \dots, x_n)$ and $y = (y_1, y_2, \dots, y_m)$. The symbols come from some alphabet $\mathcal{A}$, e.g. the four bases {A, G, C, T} for DNA or, in the case of amino acids, the 20 symbols {A, R, N, D, C, Q, E, G, H, I, L, K, M, F, P, S, T, W, Y, V}.

For now we will only consider non-gapped alignments such as:

```
HBA_HUMAN   GSAQVKGHGKKVADALTNAVAHVDDMPNALSALSDLHAHKL
            G+ +VK+HGKKV  A+++++AH+D++ +++++LS+LH  KL
HBB_HUMAN   GNPKVKAHGKKVLGAFSDGLAHLDNLKGTFATLSELHCDKL
```

Given a pair of aligned sequences (without gaps), we want to assign a score to the alignment. The two hypotheses are: The sequences are related or unrelated. The score should measure the relative likelihood that the sequences are related, *model M*, as opposed to being unrelated, *model R*, i.e.,

$$\frac{P(x, y \mid M)}{P(x, y \mid R)}.$$

The unrelated or *random model R* assumes that the letter a occurs independently with some frequency q_a, and hence the probability of the two sequences is the product:

$$P(x, y \mid R) = \prod_i q_{x_i} \prod_j q_{y_j}.$$

In the *match model M*, aligned pairs of residues occur with a joint probability p_{ab}, which is the probability that a and b are observed when they have evolved from some unknown original residue c as their common ancestor.

Thus, the probability for the whole alignment is:

$$P(x, y \mid M) = \prod_i p_{x_i y_i}.$$

The ratio of theses two likelihoods is known as the *odds ratio*:

$$\frac{P(x, y \mid M)}{P(x, y \mid R)} = \frac{\prod_i p_{x_i y_i}}{\prod_i q_{x_i} \prod_i q_{y_i}} = \prod_i \frac{p_{x_i y_i}}{q_{x_i} q_{y_i}}.$$

To obtain an additive scoring scheme, we take the logarithm to get the *log-odds ratio:*

$$S = \sum_i s(x_i, y_i),$$

with

$$s(a, b) := \log\left(\frac{p_{ab}}{q_a q_b}\right).$$

We thus obtain a matrix $s(a, b)$ that determines a score for each aligned residue pair, known as a *score* or *substitution* matrix.

For amino-acid alignments, commonly used matrices are the PAM and BLOSUM matrices, for example the BLOSUM50 matrix we have seen before.

Taking the logarithm of the likelihoods has the advantage that multiplication of probabilities is replaced with the addition of score values.

In practice the logarithmized values are usually multiplied by 10 or so and then rounded to integers.

Moreover, people often add a constant to each entry in the matrix, other than amino acid character against a space. Such a "muted" matrix has been observed to be beneficial in obtaining "better" alignments in some contexts.

Note for example, that the Smith-Waterman algorithm will only work reasonably if the "expected score value of an alignment column" is negative, i.e.,

$$\sum_{a,b \in \mathcal{A}} s(a, b) q_a q_b < 0\ ,$$

because otherwise an extremely long gapless alignment will have an expected nonnegative score simply by chance, which is clearly not what we want.

Work on substitution matrices was pioneered by Margaret Dayhoff in the 1960's. The probabilities $p_{a,b}$ and q_a were estimated from the empirical frequencies in sequences that were less than 15% different from one another. In this similarity range, the few exisiting indels could easily be spotted, and obtaining the alignments was not a problem.

Since then the methods have been refined considerably, and e.g. the book of Gusfield is recommended for further reading. (Gusfield, section 15.7.4, p. 383 ff.)

2.29 Reducing the space and time requirements

We have seen that the global and local alignment problem, with linear and with affine gap costs, can be solved by dynamic programming algorithms in space and time proportional to the product of the lengths of the input sequences.

Next we will see two approaches to mitigate these resource requirements.

2.30 Computing the score in linear space

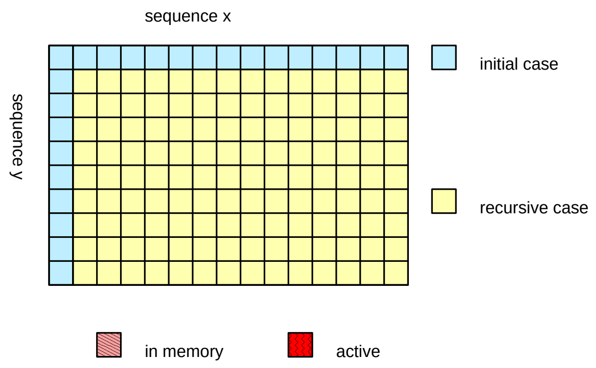
The recurrence formulas in the Needleman-Wunsch and the Smith-Watherman algorithm refer only one row and one column backward in the dynamic programming matrix.

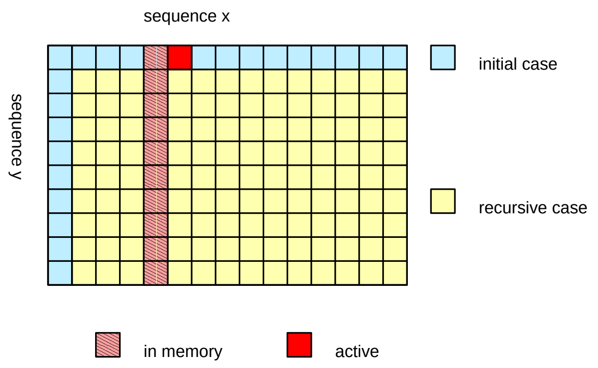
We assume that the outer loop iterates over the column index. Thus, we can carry out the computations needed for the matrix entry in the lower right corner by using only the values from *two* consecutive columns of the matrix at any given time of the execution of the algorithm.

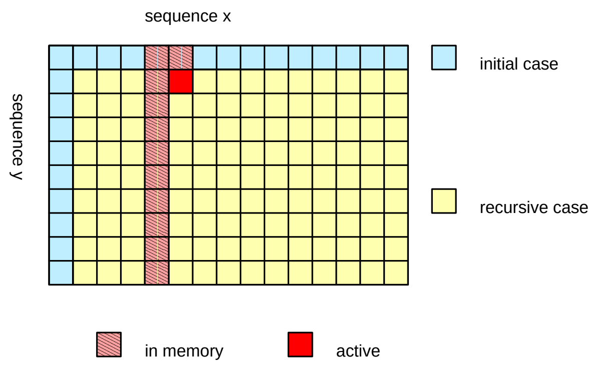
In fact, we can even come along with only *one* column and a few extra variables.

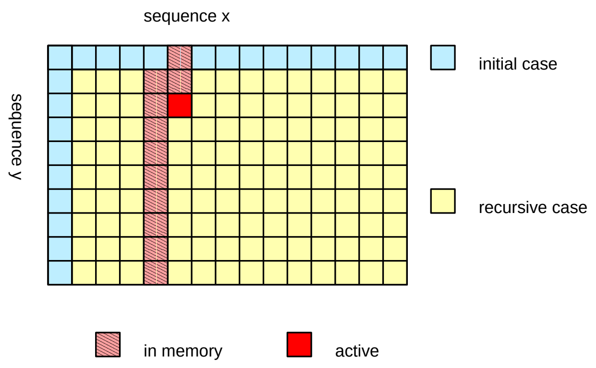
Thus far, we have distinguished "initialized" and "recursive" cases. The initialization and recursion will not change, but we will emphasize which entries of the DP matrix which are "in memory" (shaded, pink) The "active"entry (filled red) belongs to the index pair (i, j) for which the initialization or recursion is being carried out at the moment.

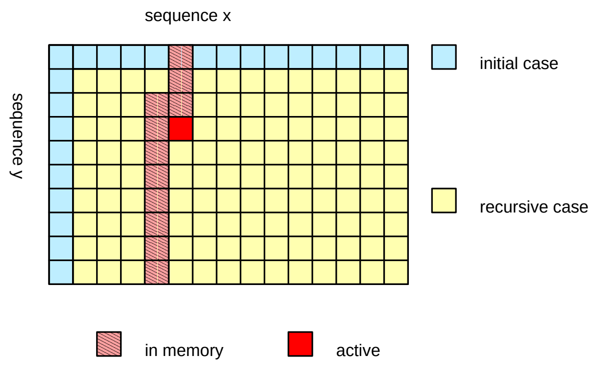
The pictures should be self-explaining . . .

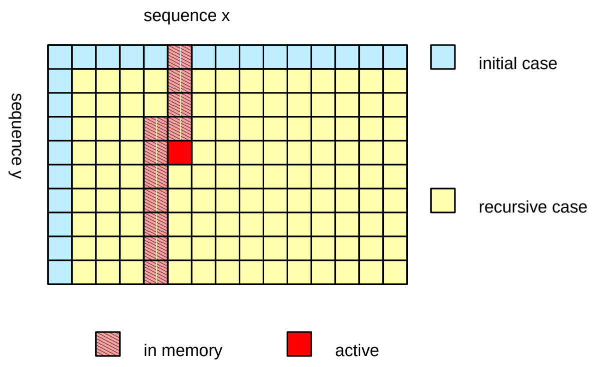


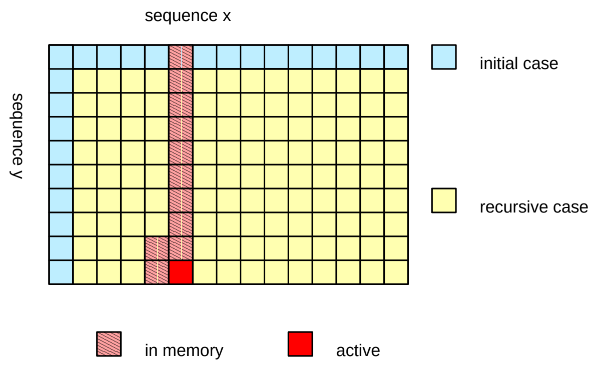












At this point ($j = n$), the calculations for the present column (with index i) of the DP matrix are finished, and we can increment the column index ($i = 0, \dots, m$) and start from the top for another round of the inner loop ($j = 0, \dots, n$).

When we have reached the cell at the lower right corner, we know the *score* of an optimal global alignment.

The algorithm still requires in $O(mn)$ time to run (where m and n are the lengths of the two sequences), but it uses only $O(\min\{m, n\})$ memory. (The min is there because we can exchange the two sequences.)

But how can we obtain the actual *alignment*, not just its score?

Paradigm: Divide et Impera

Divide et impera (lat.) = divide and conquer (engl.) = teile und herrsche (deu.)

The following observation helps. Assume this is an optimal global alignment:

```
GSAQVKGHGKKVADALTNAVAHV---D--DMPNALSALSDLHAHKL
++ ++++H+ KV   + +A  ++          +L+ L+++H+ K
NNPELQAHAGKVFKLVYEAAIQLQVTGVVVTDATLKNLGSVHVSKG
```

Then we can *split* this alignment at *any* position and will obtain two optimal global alignments for both sides.

```
GSAQVKGHGKKVADAL    |    TNAVAHV---D--DMPNALSALSDLHAHKL
++ ++++H+ KV   +    |     +A  ++          +L+ L+++H+ K
NNPELQAHAGKVFKLV    |    YEAAIQLQVTGVVVTDATLKNLGSVHVSKG
```

(Really? Exercise! For the moment, remember that we assume linear gap costs.)

The idea is to split the alignment problem recursively into smaller pieces, until they become trivial.

More generally, this is the idea of *divide and conquer*:

"Spend some effort to divide the problem into more manageable parts – then combine the final solution out of the solutions for the subproblems."

Examples of divide and conquer algorithms are: merge sort, Karatsuba multiplication, Cooley-Tuckey fast Fourier transform, ...

Sounds like a reasonable approach ... but since we do not have a global alignment yet, we do not know *where* to split the sequences.

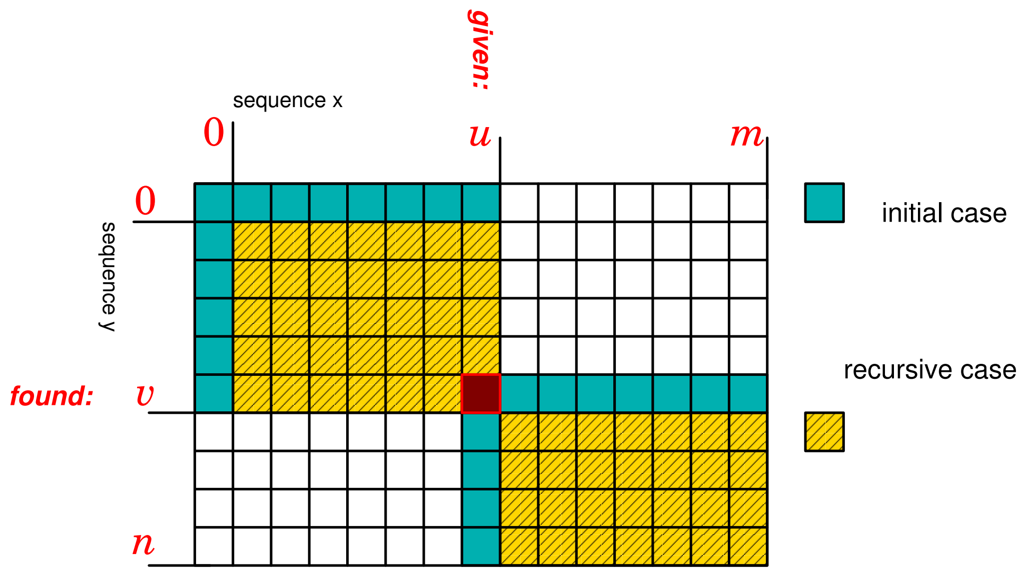
We have reduced the problem of finding an optimal alignment to another problem: *finding an optimal split position*. How can we find the optimal split position in linear space?

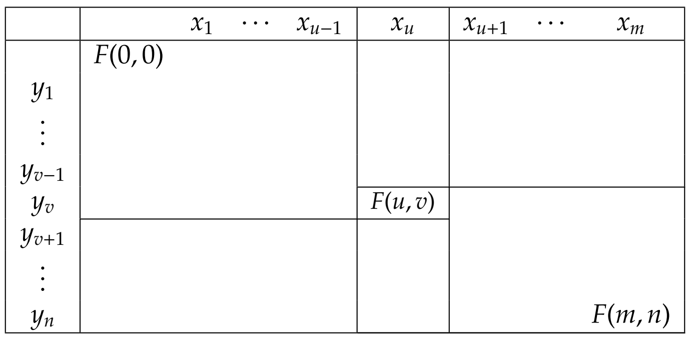
2.31 Alignment in linear space

We split the DP matrix at a *given fixed column u*. We want to split the problem into parts of about the same size, so we choose $u := \lfloor |x|/2 \rfloor$.

Then, using dynamic programming in linear space, we *find the row v* where the global alignment backtrace crosses the u-th column of the DP matrix.

Once we know that the alignment passes through the cell (u, v) of the DP matrix, we can split the problem of finding the alignment into a upper left and a lower right part.





When we have found the global alignments for both parts, we simply concatenate them.

2.32 Finding the row where to cut

The computation for the left half is done using linear space as described above. The problem with the right half is that we want to know the row v where the traceback would hit the column u.

In the setting of the NW algorithm using a full DP matrix, we could find v by tracing back from (m,n). However, this would require $\Omega(mn)$ space.

Fortunately, we do not need the entire traceback to solve our subproblem! Instead of "*incremental*" traceback pointers (the matrix T), we will maintain an array of numbers r (namely, *r*ow indices) that point *directly* into the u-th column.

For $i > u$ and any j, we *assign $r(i,j)$ to a row where a backtracking path from (i,j) crosses the u-th column*. (We say"*a* row where *a* path" because the backtracking path need not be unique.)

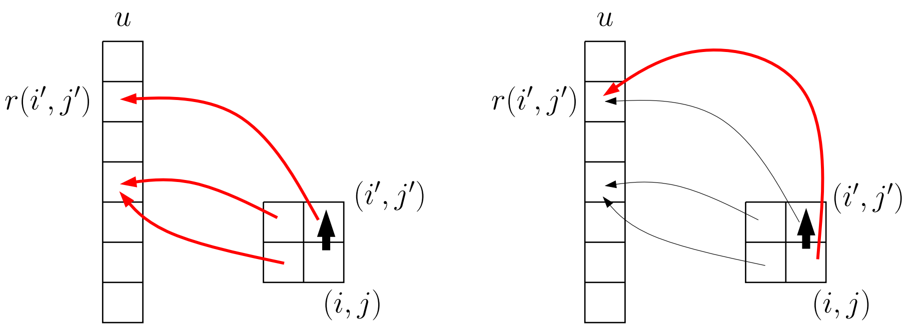
To save space, we compute the $r(i,j)$ values in linear space (using the same trick we have seen, for the score) because in the end we are only interested in one particular value: $r(m,n)$. Thus, in fact there is only an array $r(j)$ of size $m+1$, subscripted by j, but we stick to the notation $r(\cdot,\cdot)$ with two arguments for notational convenience.

We need a formula to update r. The values $r(i,j)$, for $i > u$, can be computed by the following recursion:

Let (i', j') be the cell from which $F(i, j)$ is obtained. Then

$$r(i,j) := \begin{cases} j', & \text{if } i' = u \\ r(i',j'), & \text{else} \end{cases}$$

Assume that the fat black arrow is used to assign $F(i, j)$. Then r is set as indicated.



Since $r(i, j)$ is computed from values in the previous and current column, all this can be done in $O(n)$ space.

Finally, the row index we aim at is $v = r(m, n)$. Thus we know the split position (u, v).

2.33 Time complexity of linear space alignment

Now we solve both subalignments by recursive calls of the same algorithm.

Since we do not construct the backtrace at once, we need to *re*compute many $F(i, j)$ values in recursive calls.

However the total "area" (number of entries) of F which is recomputed in the next stage of subdivision is only about *half* as much; and this goes on for *its* subproblems.

We claim that the total running time is $O(mn)$.

Proof.
First note that the area of the columns u adds up to at most $m(n + 1)$ over the whole run of the algorithm, since each column is selected at most once. The same bound applies to the rows v, because there can be at most m recursive calls in total. Thus the area of the "crosses" through all subdivision cells (u, v) is bounded by $O(mn)$.

The time to compute the split position $(\lfloor m/2 \rfloor, v)$ is bounded by

$$\lfloor m/2 \rfloor n + (m - \lfloor m/2 \rfloor)n = mn.$$

For the two subproblems in the next level of the recursion this work amounts to

$$\lfloor m/2 \rfloor v + (m - \lfloor m/2 \rfloor)(n - v) = mn/2$$

if m is even. If m is odd, we count the extra column separately. The sum of these extra columns during the complete algorithm will be at most mn.

The sum of the terms over the whole run of the algorithm is bounded by

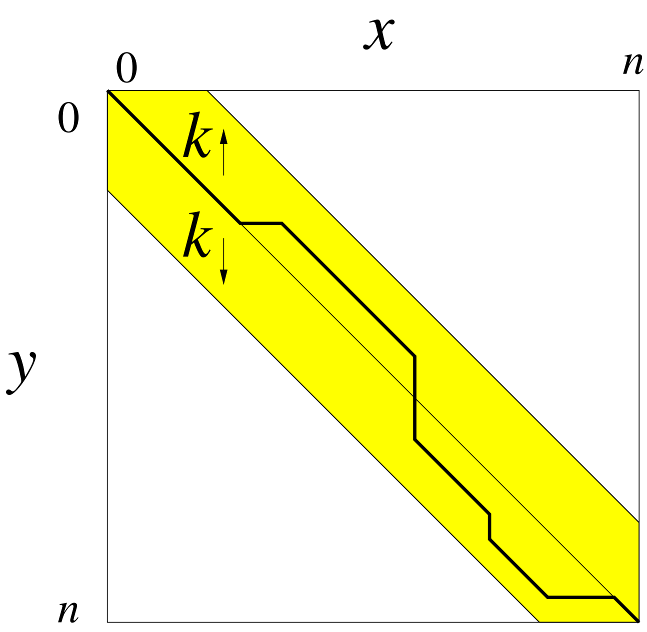
$$mn \sum_{i=0}^{\infty} 2^{-i} = 2mn = O(mn).$$

Therefore the total running time is still $O(mn)$.

2.34 Banded global alignment

Assume that we are given two sequences of equal length $n = m$, having only few differences, and we are interested in finding a global alignment with linear gap score d. In this case, it is possible to reduce the running time (and memory consumption) to linear. The ideas can be generalized to (a bit) more general settings, but we will stick to the special case for simplicity and ease of explanation.

Idea: Instead of computing the whole matrix F, we compute only a *band* of cells along the main diagonal:



Let $2k$ denote the height of the band. Obviously, the time complexity of the banded algorithm will be $O(kn)$.

The question is: Will this algorithm produce an optimal global alignment? What should k be set to?

2.35 The KBand algorithm

```
Input: two sequences $x$ and $y$ of equal length $n$, integer $k$
Output: best score $\alpha$ of global alignment using at most $k$ diagonals
away from main diagonal
Initialization: Set $F(i,0) := -id$ for all $i = 0,1,2,\ldots,k$.
Set $F(0,j) := -jd$ for all $j = 0,1,2,\ldots,k$.
for $i = 1$ to $n$ do
    for $h = -k$ to $k$ do
        $j := i + h$
        if $1 \le j \le n$ then
          $F(i,j) := F(i-1,j-1) + s(x_i,y_j)$
          if insideBand$(i-1,j,k)$ then
             $F(i,j) := \max\{F(i,j), F(i-1,j) - d\}$
          if insideBand$(i,j-1,k)$ then
        $F(i, j) := \max\{F(i, j), F(i, j-1) - d\}$
**return** $F(n, n)$
```

To test whether (i, j) is inside the band, we use:

$$\text{insideBand}(i, j, k) := (-k \le i - j \le k).$$

2.36 Searching for high-identity alignments

We can use the KBand algorithm as a fast method for finding high-identity alignments:

If we know that the two input sequences are highly similar and we have a bound b on the number of gaps that will occur in the best alignment, then the KBand algorithm with $k = b$ will compute an optimal alignment.

For example, in forensics, one must sometimes determine whether a sample of human mtDNA obtained from a victim matches a sample obtained from a relative (or from a hair brush etc). If two such sequences differ by more than a couple of base-pairs or gaps, then they are not considered a match.

2.37 Optimal alignments using KBand

Given two sequences x and y of the same length n. For simplicity, let M be a uniform match score and d the gap penalty.

Question: Let α_k be the best score obtained using the KBand algorithm for a given k. When is α_k equal to the optimal global alignment score α?

Lemma If $\alpha_k \ge M(n - k - 1) - 2(k + 1)d$, then $\alpha_k = \alpha$.

Proof. If there exists an optimal alignment with score α that does not leave the band, then clearly $\alpha_k = \alpha$. Else, all optimal alignments leave the band somewhere. This requires insertion of at least $k + 1$ gaps in each sequence, and allows only at most $n - k - 1$ matches, giving the desired bound. □

2.38 Optimal alignment using repeated KBand

The following algorithm computes an optimal alignment by repeated application of the KBand algorithm, with larger and larger k:

```
**Input:** two sequences $x$ and $y$ of the same length
**Output:** an optimal global alignment of $x$ and $y$
Initialize $k := 1$ (or some small number)
**repeat**
    compute $\alpha_k$ using KBand
    **if** $\alpha_k \ge M(n - k - 1) - 2(k + 1)d$ **then**
        **return** $\alpha_k$
        $k := 2k$
end
```

As usual, we omit details of the traceback.

2.39 Analysis of time complexity

The algorithm terminates when:

$$\begin{aligned}\alpha_k \geq M(n-k-1) - 2(k+1)d \quad &\Leftrightarrow \\ \alpha_k - Mn + M + 2d \geq -(M+2d)k \quad &\Leftrightarrow \\ -\alpha_k + Mn - (M+2d) \leq (M+2d)k \quad &\Leftrightarrow \\ \frac{Mn-\alpha_k}{M+2d} - 1 \leq k\end{aligned}$$

At this point, the total complexity is:

$$n + 2n + 4n + \cdots + kn \leq 2kn.$$

So far, this doesn't look better than nn. To bound the total complexity, we need a bound on k.

When the algorithm stops for k, we must have:

$$\frac{k}{2} < \frac{Mn - \alpha_{\frac{k}{2}}}{M+2d} - 1.$$

There are two cases: If $\alpha_{\frac{k}{2}} = \alpha_k = \alpha$, then

$$k < 2\left(\frac{Mn-\alpha}{M+2d} - 1\right).$$

Otherwise, $\alpha_{\frac{k}{2}} < \alpha_k = \alpha$. Then any optimal alignment must have more than $k/2$ spaces, and thus

$$\alpha \leq M(n - k/2 - 1) - 2(k/2+1)d \quad \Rightarrow \quad k \leq 2\left(\frac{Mn-\alpha}{M+2d} - 1\right).$$

As $M + 2d$ is a constant, it follows that k is bounded by $O(\Delta)$, with $\Delta = Mn - \alpha$, and thus the total bound is $O(\Delta n)$. □

In consequence, the more similar the sequences, the faster the KBand algorithm will run!