Exercise 2, solutions

1. Global alignment simulation.

Compute the global alignment scores $s_{ij}$ and especially $s_{mn} = S(A, B)$ for $A = \text{ACCGATG}$ and $B = \text{ACGGCTA}$ using indel penalty $-d$, where $d = 1$, and the substitution matrix:

$$
\begin{array}{|c|cccc|}
\hline
s(a, b) & 'A' & 'C' & 'G' & 'T' \\
\hline
'A' & 1 & -1 & -0.5 & -1 \\
'C' & -1 & 1 & -1 & -0.5 \\
'G' & -0.5 & -1 & 1 & -1 \\
'T' & -1 & -0.5 & -1 & 1 \\
\hline
\end{array}
$$

Trace an optimal alignment.

Rather than simulating on paper, you can also opt to implement.

Solution. The matrix with $s_{ij}$ values is presented below.

\[
\begin{array}{cccccccc}
A & C & G & G & C & T & A \\
0 & -1 & -2 & -3 & -4 & -5 & -6 & -7 \\
A & -1 & 1 & 0 & 1 & -2 & -3 & -4 & -5 \\
C & -2 & 0 & 2 & 1 & 0 & -1 & -2 & -3 \\
C & -3 & -1 & 1 & 1 & 0 & 1 & -2 & -3 \\
G & -4 & -2 & 0 & 2 & 2 & 1 & 0 & -0.5 \\
A & -5 & -3 & -1 & 1 & 1.5 & 1 & 0 & 1 \\
T & -6 & -4 & -2 & 0 & 0.5 & 1 & 2 & 1 \\
G & -7 & -5 & -3 & -1 & 1 & 0 & 1 & 1.5 \\
\end{array}
\]

The (only) optimal global alignment is therefore:

\[
\begin{array}{c}
\text{ACCGATG} \\
\mid \mid \mid \mid \\
\text{ACGGCTA}
\end{array}
\]
2. **Local alignment simulation.**

Compute the local alignment scores $l_{ij}$ for the same example as above. Trace an optimal local alignment.

**Solution.** The matrix with $l_{ij}$ values is presented below. Arrows used for backtracking are drawn only for paths originating from the maximal value in the matrix.

\[
\begin{array}{ccccccc}
 & A & C & G & G & C & T & A \\
A & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
C & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
C & 0 & 0 & 2 & 1 & 0 & 1 & 0 \\
G & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
A & 0 & 0 & 0 & 0 & 2 & 2 & 1 \\
T & 0 & 0 & 0.5 & 0 & 0.5 & 1 & 1 \\
G & 0 & 0 & 1.5 & 1 & 0 & 1 & 1.5 \\
\end{array}
\]

All local alignments with maximal score are:

\[
\begin{align*}
& AC \quad CG \quad ACCG \quad ACCG \quad ACCGAT \\
& || \quad || \quad || \quad || \quad || \quad || \quad || \\
& AC \quad CG \quad AC-G \quad ACCG \quad ACGCT \\
\end{align*}
\]

3. **Space improvement I.**

Give pseudocode (or e.g. python code) for global alignment algorithm using only space $O(m)$ to compute $S(A,B)$.

**Solution sketch.** Let $P$ be column 0 and $N$ be column 1. Initialize $P$ and compute $N$ from $P$. Copy $N$ to $P$ and compute $P$ from $N$ regarding $P$ now as column 1 and $N$ as column 2. Continue the process until $P$ has been computed as $m$-th column.

4. **Space improvement II.**

Give pseudocode (or e.g. python code) for tracing an optimal path for maximum scoring local alignment, using space quadratic in the alignment length.

**Solution sketch.** Consider a matrix $E[0..m,0..n]$ with $E[i,j] = (i',j')$, where an optimal local alignment ending at $(i,j)$ starts at $(i',j')$. During the local alignment computation, set $E[i,j] = (i,j)$ if 0 is the option taken in maximization. Otherwise set $E[i,j] = E[i*,j*]$, where $(i*,j*)$ is the cell from where the maximum score is obtained at $(i,j)$.

Now we can use the $O(m)$ space algorithm of previous assignment with the modifications i) 0 is considered in maximization, ii) filling of matrix $E$ is simulated analogously with two columns swiping through the matrix, and iii) tuple $(i',j',i,j,s)$ is recorded during computation with index pair $(i,j)$ corresponding to the maximum score $s$ in the local alignment matrix and index pair $(i',j') = E[i,j]$. 
Finally, *global* alignment matrix can be filled with inputs $A[i' + 1..i]$ and $B[j' + 1..j]$. The size of this matrix is quadratic in the alignment length and an optimal local alignment can be traced back using it.

It is easy to modify the algorithm to output a local alignment for each cell $(i, j)$ with score $s$ greater than a given threshold.

5. **Shortest detour.**

The shortest detour algorithm assumes you can fill only the diagonal zone of the dynamic programming matrix. How can you do this without allocating memory for the whole matrix $(d_{ij})$? *Hint.* Coordinate change helps; allocate a matrix only containing the diagonal zone. Find a bijective map between cells in $d_{ij}$ and cells in your new matrix.

**Solution sketch.** For the diagonal zone $[-x..n - m + x]$, one can use a matrix $D^z[0..n - m + 2x + 1, 0..m + 1]$ interpreting $(i, j)$ as $(j - i + x + 1, i)$. Initializing first and last column of $D^z$ to $\infty$, and using the coordinate change for all cells involved in the basic dynamic programming recurrence for edit distance, one can safely compute all values inside the diagonal zone.

6. **Sparse dynamic programming I.**

Next week we will study sparse dynamic programming and we exploit a *range minimum query* data structure (outside the lecture script material):

**Lemma 3.1.** The following two operations can be supported with a balanced binary search tree $T$ in time $O(\log n)$, where $n$ is the number of leaves in the tree.

- **update** $(k, \text{val})$: For the leaf $w$ with key$(w) = k$, update value$(w) = \text{val}$.
- **RMQ** $(l, r)$: Return $\min_{w : l \leq \text{key}(w) \leq r} \text{val}(w)$ (*Range Minimum Query*).

Moreover, the balanced binary search tree can be built in $O(n)$ time, given the $n$ pairs $(\text{key}, \text{value})$ sorted by component key.

Prove the lemma formally or give an example how the proposed structure works e.g. on 8 (value,key) pairs stored in its leaves and by visualizing the computation of some range minimum query for some non-empty interval.

**Proof.** Store for each internal node $v$ the minimum among value$(i)$ associated to the leaves $i$ under it. Anologously to the leaves, let us denote this minimum value value$(v)$. These values can be easily updated after a call of an operation update$(k, \text{val})$; only the $O(\log n)$ values on the path from the updated leaf towards the root need to be modified.

It is hence sufficient to show that query RMQ$(l, r)$ can be answered in $O(\log n)$ time: Find node $v$, where the search paths to keys $l$ and $r$ separate (can be the root, or empty when there is at most one key in the query interval). Let path$(v, l)$ denote the set of nodes through which the path from $v$ goes when searching for key $l$, excluding node $v$ and leaf $L$ where the search ends. Similarly, let path$(v, r)$ denote the set of nodes through which the path from $v$ goes when searching for key $r$, excluding node $v$ and leaf $R$ where the search ends. Figure 1 illustrates these concepts.

Now for each node in path$(v, l)$, where the path continues to the left, it holds that the keys $k$ in the right subtree are at least $l$ and at most $r$. Choose $vl = \min_{v' \in V'}(\text{value}(v'))$, where $V'$ is the set of roots of their right subtrees. Similarly for each node in path$(v, r)$, where the path continues to the right, it holds that the keys $k$ in the left subtree are at most $r$ and at least $l$. Choose $vr = \min_{v'' \in V''}(\text{value}(v''))$.
where \( V'' \) is the set of roots of their left subtrees. If \( L = l \) update \( vl = \min(vl, \text{value}(L)) \). If \( R = r \) update \( vr = \min(vr, \text{value}(R)) \). The final result is \( \min(vl, vr) \).

The correctness follows from the fact that the subtrees of nodes in \( V' \cup V'' \) contain all keys that belong to the interval \( [l..r] \), and only them (excluding leaves \( L \) and \( R \), which are taken into account separately). The running time is clearly \( O(\log n) \).

7. **Sparse dynamic programming II.**

A van Emde Boas tree (vEB tree) supports in \( O(\log \log n) \) time insertions, deletions, and predecessor/successor queries for values in interval \( [1..n] \). Predecessor query returns the largest element \( i' \) stored in the vEB tree smaller than query element \( i \). Successor query returns the smallest element \( i' \) stored in the vEB tree greater than query element \( i \). Show how the structure can be used instead of a balanced search tree of Lemma 3.1 to solve range minimum queries for semi-infinite intervals \( (-\infty, i] \) (i.e. for the type of queries we use e.g. in the LCS algorithm to be studied next week).

**Solution sketch.** Let \( V[k] \) store the value associated to key \( k \). Insert \( k \) to vEB tree only if predecessor of \( k \) in vEB has greater value than \( V[k] \). When inserting \( k \), delete all keys from vEB tree whose key is greater than \( k \) and value smaller than \( V[k] \). Repeat deletion of successor of \( k \) until \( V[\text{succ}(k)] < V[k] \). If key \( k \) is already in the vEB tree and its associated value \( v \) is smaller than \( V[k] \), \( V[k] \) is updated to \( v \) and the successor deletion is repeated again. Each key can be deleted only ones, so the consecutive deletions amortize in the total running time when used in the context of the LCS algorithm. The minimum value in the range \( [-\infty, i] \) is \( V[k] \) with \( k \) being the predecessor of \( i + 1 \).

8. **Space improvement III.**

Develop an algorithm for tracing an optimal path for maximum scoring local alignment, using space linear in the alignment length. **Hint.** Let \([i', i] \times [j', j]\) define the rectangle containing a local alignment. Assume you know \( j_{\text{mid}} \) for row \( (i - i')/2 \) where the optimal alignment goes through. Then you can independently recursively consider rectangles defined by \([i', (i - i')/2] \times [j', j_{\text{mid}}]\) and \([(i - i')/2, i] \times [j_{\text{mid}}, j]\). How to compute \( j_{\text{mid}} \)?

**Solution sketch.** To find \( j_{\text{mid}} \) use an algorithm analogous to assignment 3. Let \( m = i - i' \), \( n = j - j' \), \( A' = A[i' + 1..i] \), and \( B' = B[j' + 1..j] \). Redefine \( E[i, j] = j_{\text{mid}} \) for \( i \geq m/2 \). To compute its content,
do as follows. While computing the *global* alignment of $A'$ and $B'$, set $E[m/2, j] = j$ at row $m/2$, and at any row after $m/2$ follow the same logic as before assigning $E[i, j] = E[i*, j*]$ with $(i*, j*)$ being the cell where the maximum was taken in the global alignment computation. Then $E[m, n]$ finally contains the requested value $j_{mid}$ such that an optimal path to $(m, n)$ goes through $(m/2, j_{mid})$. This algorithm can be simulated using the two column approach in space $O(m)$.

Then one can follow as hinted, repeating this same algorithm recursively. The number of cells computed is $mn + mn/2 + mn/4 + \cdots + n < 2mn$. At each subrectangle, the space used is bounded by $O(m)$.

This algorithm is known as the Hirschberg’s algorithm.