Exercise 3, solutions (selected)

2. Database search II.

Implement the $k$-mer index: write a program that gets value $k$ and a sequence $S$ as an input and produces for every $k$-mer $W$ a list that contains pointers (indexes) to the occurrence locations of $W$ in $S$. Hint. This can be done, e.g., by modifying the code for $k$-th order Markov chain from Problem 8 of Exercise 1. Basic version is enough without considering how to compress the lists of occurrences.

Solution.

3. Database search III.

Implement BLAST-like search on top of the $k$-mer index of the previous assignment to report candidate occurrences. You can ignore the dynamic programming part.

Solution.

5. Affine gap scores I.

The lecture material defines gaps as runs of indels. Consider the alternative definition of gaps as runs of insertions or runs of deletions. Modify the basic recurrence for alignment under affine gap scores accordingly. What running time you obtain?

Solution.
Let $s_{a_{i,j}}$ denote the optimal score of aligning $A_{1..i}$ and $B_{1..j}$ under affine gap score model. Observe that this differs from $s_{m_{i,j}}$, that gives the optimal score among those ending with a match. The recurrence becomes

$$s_{a_{i,j}} = \max \{ s_{a_{i-1,j-1}} + s(a_i, b_j), \max_{0 \leq i' < i} (s_{a_{i',j}} - \alpha - \beta (i - i' - 1)), \max_{0 \leq j' < j} (s_{a_{i,j'}} - \alpha - \beta (j - j' - 1)) \}.$$

with initialization:

$$s_{a_{0,0}} = 0,$$
$$s_{a_{i,0}} = -\alpha - (i - 1)\beta, \text{ for all } 1 \leq i \leq m,$$
$$s_{a_{0,j}} = -\alpha - (j - 1)\beta, \text{ for all } 1 \leq j \leq n,$$

It takes $O(mn(m + n))$ time to evaluate.

Note: a default assumption about $\alpha$ and $\beta$ is that $\alpha \gg \beta$. A consequence of this assumption is that in the optimal alignment it is never beneficial (in order to maximize the score) to have two adjacent gaps of the same type (for example, a run of deletions followed by another run of deletions). It may seem at first that the above recurrence gives the optimal score for any values $\alpha$ and $\beta$ (in particular, also when the assumption $\alpha \gg \beta$ does not hold) because the recurrence for $s_{a_{i,j}}$ allows the case of two adjacent gaps of the same type. This is however not true (i.e., the above algorithm still requires $\alpha \gg \beta$), because of boundary case initialization, i.e., we assume that the score of the optimal alignment of $A_{1..i}$ and empty string (see $s_{a_{i,0}}$) is $-\alpha - (i - 1)\beta$, whereas for arbitrary values of $\alpha$ and $\beta$ a higher score may be achieved by splitting a single gap of deletions into two or more gaps. Of course for symmetric reason the initialization $s_{a_{0,j}} = -\alpha - (j - 1)\beta$ also requires $\alpha \gg \beta$. 

1
6. **Affine gap scores II.**

Consider the setting above. Modify the Gotoh algorithm to handle the runs of insertions and runs of deletions separately. *Hint.* You will need three tables.

**Solution.**

By \( \text{sm}_{i,j} \), \( \text{si}_{i,j} \), and \( \text{sd}_{i,j} \) we denote the optimal score for \( A_{1..i} \) and \( B_{1..j} \) ending in match, insertion, and deletion (respectively). Then, the following recurrences hold for \( 1 \leq i \leq m \) and \( 1 \leq j \leq n \):

\[
\text{sm}_{i,j} = \max\{\text{sm}_{i-1,j-1}, \text{si}_{i-1,j-1}, \text{sd}_{i-1,j-1}\} + s(a_i, b_j),
\]

\[
\text{si}_{i,j} = \max\{\text{si}_{i,j-1} - \beta, \text{sm}_{i,j-1} - \alpha, \text{sd}_{i,j-1} - \alpha\},
\]

\[
\text{sd}_{i,j} = \max\{\text{sd}_{i-1,j} - \beta, \text{sm}_{i-1,j} - \alpha, \text{si}_{i-1,j} - \alpha\}.
\]

All arrays are initialized as follows:

\[
\begin{align*}
\text{sm}_{0,0} &= 0, \\
\text{sm}_{i,0} &= -\infty, \text{ for all } 1 \leq i \leq m, \\
\text{sm}_{0,j} &= -\infty, \text{ for all } 1 \leq j \leq n, \\
\text{si}_{0,0} &= 0, \\
\text{si}_{i,0} &= -\infty, \text{ for all } 1 \leq i \leq m, \\
\text{si}_{0,j} &= -\alpha - \beta(j - 1), \text{ for all } 1 \leq j \leq n, \\
\text{sd}_{0,0} &= 0, \\
\text{sd}_{i,0} &= -\alpha - \beta(i - 1), \text{ for all } 1 \leq i \leq m, \\
\text{sd}_{0,j} &= -\infty, \text{ for all } 1 \leq j \leq n.
\end{align*}
\]

We then have \( S_G(A_{1..i}, B_{1..j}) = \max(\text{sm}_{i,j}, \text{si}_{i,j}, \text{sd}_{i,j}) \).

Note: as in the previous problem, the above solution requires \( \alpha \gg \beta \).

7. **Affine gap scores III.**

Consider the setting above. Modify the invariant-based algorithm to handle runs of insertions and runs of deletions separately.

**Solution.**

Consider the solution to problem 5. Consider filling a matrix column-by-column and reaching cell \((i, j)\). If you have maintained \( r_i = \max_{0 \leq i' < i}\{\text{sa}_{i',j} + \beta i'\} \) and \( c_j = \max_{0 \leq j' < j}\{\text{sa}_{i,j'} + \beta j'\} \), it suffices to update

\[
\text{sa}_{i,j} = \max\{\text{sa}_{i-1,j-1} + s(a_i, b_j), \begin{align*}
& r_i - \alpha - \beta(i - 1), \\
& c_j - \alpha - \beta(j - 1), \end{align*}\}
\]

set \( r_i = \max(r_i, \text{sa}_{i,j} + \beta i) \) and \( c_j = \max(c_j, \text{sa}_{i,j} + \beta j) \).

This takes \( O(mn) \) time to evaluate.
Extra problems

1. Sparse dynamic programming I.

Show that set \( M = M(A, B) = \{(i,j) : a_i = b_j\} \) needed for sparse dynamic programming LCS computation, sorted in reverse column-order, can be constructed in time \( O(\sigma + |A| + |B| + |M|) \) time on constant alphabet and in \( O(|B| + |M| \log |A|) \) time on ordered alphabet. Observe also that this construction can be run in parallel with the main algorithm to improve the space requirement to \( O(m) \).

Solution.

First assume constant alphabet. Let \( P[c] = \{i : a_i = c\} \) for \( c = 0, \ldots, \sigma - 1 \). The array needs \( O(\sigma + |A|) \) space and can easily be computed from \( A \) in \( O(\sigma + |A|) \) time. Given \( P \) it is easy to compute \( M \): iterate though \( B \) and for every \( j = 1, \ldots, |B| \) add to \( M \) all pairs \((i,j)\) such that \( i \in P[b_j] \). This algorithm needs \( O(\sigma + |A| + |B| + |M|) \) time in total and requires \( O(\sigma + |A| + |M|) \) space.

To generalize the above solution for ordered alphabet, we replace the array \( P \) with an array \( P' \) containing all pairs \((a_i,i), i = 1, \ldots, |A|\), sorted by the first element (and in the case of ties – by the second element). Computing \( P' \) takes \( O(|A| \log |A|) \) time. Given \( P' \), we compute \( M \) using the same method as above, except to enumerate all \( i \) such that \( a_i = b_j \) for a given index \( j \), we first compute the range \([i',i'']\) in \( P' \) containing all pairs having \( b_j \) as the first component using two binary searches, and then simply scan \( P'[i',i''] \) to obtain the indexes from the second components of each pair in the range. The time for performing binary searches over the whole algorithm is \( O(|B| \log |A|) \), and all other operations take \( O(|M|) \) time. The resulting algorithm thus has the time complexity \( O(|M| + (|A| + |B|) \log |A|) \), which is \( O(|M| + |B| \log |A|) \) (and in particular also runs in the time bound from the problem statement) if we assume \(|A| \leq |B|\) and uses \( O(|A| + |M|) \) space.

To reduce the space usage to \( O(|A|) = O(m) \) it suffices to observe that in both algorithms above, the set \( M \) is constructed in the column-order. However, if we modify the algorithm to enumerate all \( i \) such that \( a_i = b_j \) for a given \( j \) in the decreasing order (which is very easy to accomplish, e.g., in the second algorithm we scan \( P'[i',i''] \) right-to-left instead of left-to-right), we obtain \( M \) in the reverse column-order, which is exactly the order in which we access \( M \) in the LCS algorithm. Since we only access each element of \( M \) once, we don’t need to ever store whole \( M \) but only \( P \) or \( P' \), either of which takes \( O(m) \) space.

2. Sparse dynamic programming II.

The sparse dynamic algorithm for distance \( D_{id}(A, B) \) can be simplified significantly if derived directly for computing \( |LCS(A, B)| \). Derive this algorithm. Hint. The search tree can be modified to answer range maximum queries instead of minimum.

Solution. The simplified algorithm is given below.

---

**Algorithm 1:** Sparse dynamic programming for \( |LCS(A, B)| \).

**Input:** Sequences \( A = a_1a_2\ldots a_m \) and \( B = b_1b_2\ldots b_n \).

**Output:** \( |LCS(A, B)| \).

1. Construct set \( M = \{(i,j) : a_i = b_j\} \) and sort it into an array \( M[1..|M|] \) in reverse column-order;
2. Initialize a search tree \( T \) with keys \( 0, 1, \ldots, m \);
3. \( T.update(0, 0) \);
4. for \( p \leftarrow 1 \) to \(|M|\) do
5. \( (i,j) \leftarrow M[p] \);
6. \( d \leftarrow 1 + T.RMaxQ(0, i - 1) \);
7. \( T.update(i, d) \);
8. return \( T.RMaxQ(0, m) \);