

Introduction to Computational Biology
Homework 2
Solution

Problem 1: Number of alignments

We discussed in class that the number of alignments of two sequences x and y is exponential in their length. This problem is designed to prove this fact.

We will count only the number of distinct alignments. Of course we need to define what distinct really means. For this, define an equivalence on alignments as follows: Two alignments are equivalent if they align the same indices of x to the same indices of y .

For instance, the following two alignments are equivalent:

```
A-GCTTT-GA
-AG-T--CG-

-AGCT-TTGA
A-G-TC--G-
```

because they align x_2 with y_2 , x_4 with y_3 , and x_7 with y_5 .

It follows that two alignments are distinct if they are not equivalent.

Let $|x| = m$ and $|y| = n$.

(a) Show that the number of distinct alignments is $\binom{m+n}{n}$.

Solution: Let C_k be the number of distinct alignments that match k symbols of x to k symbols of y . Then if $n \leq m$, the total number of distinct alignments is simply $\sum_{k=1}^n C_k$.

C_k can be computed as the number of possible ways of choosing k symbols from x and k symbols from y . Therefore,

$$C_k = \binom{n}{k} \binom{m}{k} = \binom{n}{n-k} \binom{m}{k}$$

The last equality follows from the fact that “taking” k from n is the same as “throwing” $n - k$ from n . Mathematically speaking, $\binom{n}{k} = \binom{n}{n-k}$.

Now the number of distinct alignments is

$$\sum_{k=1}^n \binom{n}{n-k} \binom{m}{k}$$

This is essentially all possible ways of choosing n from $n + m$. Therefore, the number of distinct alignments is $\binom{m+n}{n}$

(b) For $m = n$, use Stirling formula $n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$ to prove that the number of distinct alignments is approximately $\frac{2^n}{\sqrt{\pi n}}$

Solution: Obvious when we replace $n!$ by its approximation in the expression $\binom{2n}{n}$

(c) Can two sequences x and y have an exponential number of distinct *optimal* alignments? If yes, construct two such sequences. If no, prove it.

Solution: Yes they can. An example would be the sequence $x = AAA\dots A$ of length m and the sequence $y = AA\dots A$ of length n where $m = 2n$. The number of distinct alignments that align n symbols of y to n symbols of m is $\binom{2n}{n}$, and all these alignments are optimal. This is a term exponential in n as shown above.

(d) Suggest an algorithm that produces all optimal alignments in $O(mn + L(m + n))$ where L is the number of optimal alignments.

Solution: We compute the dynamic programming table with the trace back pointers. This takes $O(mn)$ time. From $A(m, n)$, if we trace back on a path to $A(0, 0)$ we obtain an alignment. This takes $O(m + n)$ time which is the maximum length of an alignment. If we trace back on all paths, then we obtain all alignments. So the total running time would be $O(mn + L(m + n))$. The question becomes how to trace back on every path from $A(m, n)$ to $A(0, 0)$. We do this recursively.

We can abstract the dynamic programming table with the back trace pointers as a directed graph where $A(i, j)$ represents a node, and a pointer represents a directed edge. Assume the existence of a function called *append* that given an alignment starting at $A(i, j)$ and given an $A(k, l) \in \text{adj}[A(i, j)]$ it appends to the beginning of the alignment, the column corresponding to an edge from $A(i, j)$ to $A(k, l)$.

```

a ← ∅
visit(A(m, n), a)

```

```

visit(A(i, j), a)
if adj[A(i, j)] = ∅    (i.e. A(0, 0))
  then output a
  else for every A(k, l) ∈ adj[A(i, j)]
    do visit(A(k, l), append(a, (A(i, j), A(k, l))))

```

The approach above consumes $O(L(m+n))$ space because each alignment has to be stored at every step of the recursion. A better way is to have only one shared copy of a (as opposed to passing it as a parameter) that is updated appropriately at each level of the recursion. This consumes only $O(m+n)$ space. Yet another way is the following:

For every edge e in the directed graph, we associate $count(e)$ equal to the number of paths from $A(m, n)$ to $A(0, 0)$ that use edge e . Starting from $A(m, n)$ we follow an arbitrary path to $A(0, 0)$ and decrement $count(e)$ for every edge e we pick on the path. When we reach $A(0, 0)$ we obtain an alignment that we can output. We repeat this until every edge incident to $A(m, n)$ has zero count. At this point, we definitely outputted all the optimal alignments.

How to compute $count(e)$ for every edge e ? We define $b(A(i, j))$ to be the number of ways $A(i, j)$ can be reached from $A(m, n)$. We start with $b(A(m, n)) = 1$ and in a backward rowwise manner we compute $b(A(i, j))$ as the sum of all $b(A(k, l))$ such that there is an edge from $A(k, l)$ to $A(i, j)$.

We reverse the direction of all edges in the graph and repeat the same thing starting from $A(0, 0)$ in a forward rowwise manner, to obtain, for every $A(i, j)$, $f(A(i, j))$, the number of ways $A(i, j)$ can be reached from $A(0, 0)$.

Now for an edge $e = (A(i, j), A(k, l))$, it can be seen that $count(e) = b(A(i, j)) \cdot f(A(k, l))$. Determining $count(e)$ for every e in this way takes only $O(mn)$ time and space.

Problem 2: Linear space alignments

Let x and y be two sequences with $|x| = m$ and $|y| = n$.

(a) Describe a linear space $O(mn)$ time algorithm to compute the optimal semi-global alignment between x and y .

Solution: We can reduce the problem of obtaining the optimal semi-global alignment to the problem of obtaining an optimal local alignment and then use part (b). Here's how: modify x by adding a special character at the beginning and at the end, e.g. $x' = \$x\$$. Modify y similarly, i.e. $y' = \$y\$$. Modify the scoring scheme such that matching $\$$ with a gap contributes 0 and

matching \$ with anything other than gap contributes a large enough score, e.g. $> \max(m, n)[\max_{a,b} s(a, b) - \min_{a,b} s(a, b)]$ (the min is negative). Now find the optimal local alignment of x' and y' . This local alignment has to match at least two \$ symbols (either one from each sequence, or two in the same sequence). Otherwise, we can get a better score by just considering a local alignment that matches a \$ in x' and a \$ in y' . Since \$ characters are only at the extremity, this corresponds to a semi-global alignment of x and y if we ignore the \$ characters.

(b) Describe a linear space $O(mn)$ time algorithm to compute the optimal local alignment between x and y . *Hint:* Find the two highest scoring substrings in linear space and $O(mn)$ time, then compute their optimal global alignment in linear space and $O(mn)$ time.

Solution: We will obtain in linear space the two highest scoring substrings. Then we can obtain the local alignment by performing a global alignment of these two substrings (in linear space).

First we will find r_x and r_y such that $x_{l_x} \dots x_{r_x}$ and $y_{l_y} \dots y_{r_y}$ are two highest scoring substrings. We perform the local alignment dynamic programming algorithm as before keeping one row at a time (linear space) of the matrix A . We also update r_x and r_y that correspond to the maximum entry in A as follows: r_x and r_y are initialized to zeros. If for some entry $A(i, j)$, $A(i, j) > A(r_x, r_y)$ then we update $r_x = i$ and $r_y = j$. If $A(i, j) = A(r_x, r_y)$ and $i < r_x$, we update $r_x = i$ and $r_y = j$. If $A(i, j) = A(r_x, r_y)$ and $i = r_x$ and $j < r_y$, then we update $r_x = i$ and $r_y = j$. By the end of the dynamic programming algorithm r_x and r_y correspond to the ends of some highest scoring substrings $x_{l_x} \dots x_{r_x}$ and $y_{l_y} \dots y_{r_y}$.

Now we consider the two strings $x_1 \dots x_{r_x}$ and $y_1 \dots y_{r_y}$ and reverse them (i.e. $x_{r_x} \dots x_1$ and $y_{r_y} \dots y_1$) and repeat the above dynamic programming algorithm on a matrix A' to obtain r'_x and r'_y . Now r'_x and r'_y correspond to indices in the strings $x_{r_x} \dots x_1$ and $y_{r_y} \dots y_1$ respectively. To obtain the corresponding indices in the original strings, we consider $l_x = r_x - r'_x + 1$ and $l_y = r_y - r'_y + 1$. We now claim that $x_{l_x} \dots x_{r_x}$ and $y_{l_y} \dots y_{r_y}$ are two highest scoring substrings. Since $A'(r'_x, r'_y)$ is maximum, there must be two substrings $x_{l_x} \dots x_i$ and $y_{l_y} \dots y_j$ that score optimally. Note that we cannot have $i < r_x$; otherwise, $A(i, j) = A(r_x, r_y)$ and $i < r_x$, a contradiction to how r_x was previously obtained. Therefore $i = r_x$. Similarly, we cannot have $j < r_y$; otherwise $A(r_x, j) = A(r_x, r_y)$ and $j < r_y$, a contradiction to how r_y was previously obtained. Therefore, $j = r_y$.

(c) Describe a linear space $O(mn)$ time algorithm to compute the optimal global alignment between x and y under an affine gap penalty function. (this is not hard, but can be messy, so it is optional).

Solution: With an affine gap penalty, we need three matrices A , B , and C corresponding to three types of alignments as explained in class. So in the

linear space version of the algorithm, we have to compute in each level of the recurrence a row for A , a row for A' , a row for B , a row for B' , a row for C , and a row for C' . This can still be done in linear space. These six rows correspond to the scores of aligning:

- $x_1 \dots x_{m/2-1}$ with all prefixes of y and ending with no gaps
- $x_{m/2+1} \dots x_m$ with all suffixes of y and ending with no gaps
- $x_1 \dots x_{m/2-1}$ with all prefixes of y and ending with a gap in y
- $x_{m/2+1} \dots x_m$ with all suffixes of y and ending with a gap in y
- $x_1 \dots x_{m/2-1}$ with all prefixes of y and ending with a gap in x
- $x_{m/2+1} \dots x_m$ with all suffixes of y and ending with a gap in x

Let:

- $a(j)$ = score of optimally aligning $x_1 \dots x_{m/2-1}$ with $y_1 \dots y_j$ and ending with no gap
- $a'(j)$ = score of optimally aligning $x_m \dots x_{m/2+1}$ with $y_n \dots y_j$ and ending with no gap
- $b(j)$ = score of optimally aligning $x_1 \dots x_{m/2-1}$ with $y_1 \dots y_j$ and ending with a gap in y
- $b'(j)$ = score of optimally aligning $x_m \dots x_{m/2+1}$ with $y_n \dots y_j$ and ending with a gap in y
- $c(j)$ = score of optimally aligning $x_1 \dots x_{m/2-1}$ with $y_1 \dots y_j$ and ending with a gap in x
- $c'(j)$ = score of optimally aligning $x_m \dots x_{m/2+1}$ with $y_n \dots y_j$ and ending with a gap in x

Every one of the above quantities can be obtained in constant time from the six rows.

Now we need to identify where $x_{m/2}$ should go. As before we will try aligning $x_{m/2}$ with all y_j and all gaps between y_j and y_{j+1} , and choose the best among all these. But we have to be careful on how to compute the score in each case. If we align $x_{m/2}$ with y_j , then the score is

$$s(x_{m/2}, y_j) + \max(a(j-1), b(j-1), c(j-1)) + \max(a'(j+1), b'(j+1), c'(j+1))$$

If we align $x_{m/2}$ with a gap between y_j and y_{j+1} , then the score is

$$\max \begin{cases} a(j) - e + a'(j+1) \\ a(j) - d + b'(j+1) \\ a(j) - e + c'(j+1) \\ b(j) - d + a'(j+1) \\ b(j) + e - 2d + b'(j+1) \\ b(j) - d + c'(j+1) \\ c(j) - e + a'(j+1) \\ c(j) - d + b'(j+1) \\ c(j) - e + c'(j+1) \end{cases}$$

In other words, if non of the side alignments end with a gap in y (note that the right alignment aligns suffixes so the end gap is adjacent to $x_{m/2}$), aligning $x_{m/2}$ with a gap is penalized by $-e$ (gap of size 1 in the middle). If only one of the side alignments ends with a gap in y , aligning $x_{m/2}$ with a gap is penalized by $-d$ (the opening of the gap has been taken care of). If both alignments end with a gap in y , aligning $x_{m/2}$ with a gap is penalized by $-d + (e - d) = e - 2d$. This is because the opening of a gap is counted twice (one in each side alignment), so we adjust by adding the term $e - d$.

Problem 3: Concave gap penalty function (optional for a better understanding of gap functions)

Let γ be a gap penalty function defined over non-negative integers. The function γ is called sub-additive iff it satisfies the following: $\gamma(k_1 + k_2 + \dots + k_n) \leq \gamma(k_1) + \gamma(k_2) + \dots + \gamma(k_n)$.

(a) Show that a concave γ , i.e. one that satisfies $\gamma(x+1) - \gamma(x) \leq \gamma(x) - \gamma(x-1)$, is sub-additive if $\gamma(0) \geq 0$.

Solution: We use the fact that γ is a gap function, and hence it is defined on intergers ≥ 0 only. In fact, the statement above is not true otherwise. Before we prove the statement, we give some examples when the statement is not true to justify the need for the condition above.

First, consider the function defined as follows: $\gamma(-1) = -1$, $\gamma(0) = 0$, $\gamma(1) = 1$, $\gamma(2) = 1$. This satisfies the concavity condition. Now $\gamma(2 + (-1)) = \gamma(1) = 1 > \gamma(2) + \gamma(-1) = 1 - 1 = 0$. Therefore, we cannot have γ defined on negative integers.

Second, consider the periodic function that start at zero and remains zero until $x = 1/3$, then goes up linearly from zero to 1 at $x = 2/3$, then goes down linearly to zero at $x = 1$. This is repeated for every interval of length 1. This function satisfies the concavity condition because $\gamma(x) = \gamma(x + 1)$. Now $\gamma(1/3 + \epsilon)$ is strictly positive. But $\gamma(1/3) = \gamma(\epsilon) = 0$. Therefore, $\gamma(1/3 + \epsilon) > \gamma(1/3) + \gamma(\epsilon)$. Hence, γ cannot be defined on non-integers.

Now we prove the result for γ defined on integers ≥ 0 . We prove the statement for $\gamma(k_1 + k_2)$. The general case can be easily obtained by induction.

$$\begin{aligned}
 \gamma(k_1 + k_2) &= \gamma(k_1) + \sum_{i=1}^{k_2} \gamma(k_1 + i) - \gamma(k_1 + i - 1) \\
 &\leq \gamma(k_1) + \sum_{i=1}^{k_2} \gamma(i) - \gamma(i - 1) \text{ by concavity condition} \\
 &= \gamma(k_1) + \gamma(1) - \gamma(0) + \gamma(2) - \gamma(1) + \dots + \gamma(k_2 - 1) - \gamma(k_2 - 2) + \gamma(k_2) - \gamma(k_2 - 1) \\
 &= \gamma(k_1) + \gamma(k_2) - \gamma(0) \\
 &\leq \gamma(k_1) + \gamma(k_2) \text{ if } \gamma(0) \geq 0
 \end{aligned}$$

Note that the first equality and second inequality are valid only if both k_1 and k_2 are ≥ 0 .

The next set of questions are *intended* to help you understand why the DP algorithm we saw in class requires γ to be concave. Here's the algorithm again:

$$A(i, j) = \max \begin{cases} A(i - 1, j - 1) + s(i, j) & (1) \\ A(i, j - k) - \gamma(k), \quad k = 1 \dots j & (2) \\ A(i - k, j) - \gamma(k), \quad k = 1 \dots i & (3) \end{cases}$$

Without loss of generality, we restrict our attention to alignments that end with a gap in x . Call such an alignment of $x_1 \dots x_i$ and $y_1 \dots y_j$ “good” if it ends with a gap of length k in x for some $k > 0$ and **optimally** aligns $x_1 \dots x_i$ to $y_1 \dots y_{j-k}$.

(b) Show that a “good” alignment of $x_1 \dots x_i$ and $y_1 \dots y_j$ is not necessarily optimal.

Solution: Consider the following alignment:

```
-A-
AAA
```

This alignment is “good” because -A aligned with AA is optimal (A- with AA is the only other way and is the same). However, the alignment (-A-, AAA) is not necessarily optimal, depending on γ .

(c) Show that if γ is concave, then an optimal alignment (that ends with a gap in x) of $x_1 \dots x_i$ and $y_1 \dots y_j$ is a “good” alignment.

Solution: Consider an optimal alignment that ends with a gap of length $l_1 > 0$ in x . The score of this alignment is equal to $S = S_1 - \gamma(l_1)$, where S_1 is the score of the portion of the alignment obtained by excluding the gap of length l_1 . Assume that the alignment is not “good”, i.e. S_1 is not optimal. Then there must be another alignment A for that portion with score $S_2 > S_1$. This alignment ends in a gap of length $l_2 \geq 0$ in x . Therefore, $S_2 = S_3 - \gamma(l_2)$ (assume the convention $\gamma(0) = 0$), where S_3 is the score of the portion of the alignment A obtained by excluding the gap of length l_2 . We will construct a new alignment by concatenating A with the gap of length l_1 in x . Let's find

the core of this alignment. The score is $S' = S_3 - \gamma(l_2 + l_1)$. Using part (a), $S' = S_3 - \gamma(l_2 + l_1) \geq S_3 - \gamma(l_2) - \gamma(l_1) = S_2 - \gamma(l_1) > S_1 - \gamma(l_1) = S$. Hence $S' > S$ contradicting that our alignment was optimal. Therefore, it must have been a “good” alignment too.

(d) Show that if γ is concave, then for any given alignment of $x_1 \dots x_i$ and $y_1 \dots y_j$ with score S , if we split the alignment in two parts with scores S_1 and S_2 , then $S_1 + S_2 \leq S$.

Solution: Assume the first part ends with a gap of length $l_1 \geq 0$ and the second part start with a gap $l_2 \geq 0$. Then $S_1 = A - \gamma(l_1)$ and $S_2 = -\gamma(l_2) + B$. By the sub-additive property, $S = A - \gamma(l_1 + l_2) + B \geq A - \gamma(l_1) - \gamma(l_2) + B = S_1 + S_2$.

(e) Based on parts (b) and (c) and (d), explain why step (2) in the algorithm above is both **necessary**, i.e. it must check all $k = 1 \dots j$, and **correct**, i.e. it computes the optimal alignment of $x_1 \dots x_i$ and $y_1 \dots y_j$ that ends with a gap in x .

Solution:

necessity: Step (2) should find the optimal alignment that ends in a gap in x . This gap may have any length k . Moreover, the alignment must be “good” (part c). Therefore, the score must be the sum of the score of some optimal alignment and a gap of length k . But if such a “good” alignment is found, it may not be optimal (part b); therefore, all “good” alignment must be checked.

correctness: The optimal alignment that ends in a gap in x must exist and it is “good” (part c). Therefore, step (2) will find a k such that $A(i, j - k)$ is optimal and corresponds to an alignment that ends with no gaps, and $A(i, j - k) - \gamma(k)$ is the score of the optimal alignment that ends with a gap in x (of length k). Since $S_1 + S_2 \leq S$ for any split of an alignment (part d), step (2) is guaranteed not to compute any score greater than $A(i, j - k) - \gamma(k)$.

(f) Construct an instance of an optimal alignment (that ends with a gap in x) that is not “good”. (you cannot use a concave gap function according to part (c)). Argue that the algorithm above will not work properly if such an instance can be constructed.

Solution: Assume $\gamma(0) = 0$, $\gamma(1) = 1$, $\gamma(2) = 5$. Also assume that the score of a match is +1 and the score of a mismatch is -1. Consider the following optimal alignment with score $-1 - 1 - 1 = -3$.

-A-
ABA

Any other alignment will have a score of $+1 - 5 = -4$.

In the alignment above, the alignment (-A, AB) is not optimal. (-A, AB) has a score of $-1 - 1 = -2$. The alignment (A-,AB) has a score of $1 - 1 = 0$. Therefore, we have an optimal alignment that is not “good”.

Assume that the optimal alignment of $x_1 \dots x_i$ and $y_1 \dots y_j$ ends with a gap of length k in x and it is “good”. This optimal alignment corresponds to aligning $y_1 \dots y_{j-k}$ with $x_1 \dots x_i$ (non-optimally) and aligning $y_{j-k+1} \dots y_j$ with a gap of length k . If the algorithm does not compute the correct value for $A(i, j - k)$, then we are done because it will not perform correctly on the instance $x_1 \dots x_j$ and $y_1 \dots y_{j-k}$. If the algorithm computes the correct value for $A(i, j - k)$, then step (2) will compute at some point in time $A(i, j) = A(i, j - k) - \gamma(k)$ (or a larger value) which is greater than the score of the optimal alignment. Since $A(i, j)$ never decreases, it will hold the wrong score forever, and thus the algorithm fails on the instance $x_1 \dots x_i$ and $y_1 \dots y_j$.

Problem 4: Edit distance and beyond

The edit distance of two strings x and y is defined as the minimum number of operations needed to change x into y , where each operation can be an insertion, a deletion, or a substitution.

(a) Devise a scoring scheme such that the score of the optimal alignment can be easily related to the edit distance.

Solution: Since we need to minimize the number of insertions, deletions, and substitutions, we can adopt the scoring scheme $\{0, -1, -1\}$. The score of the optimal alignment is the negative of the edit distance. The optimal alignment itself determines how x and y can be transformed into each other with minimum number of insertions, deletions, and substitutions.

(b) Devise a scoring scheme such that the score of the optimal alignment can be easily related to the length of the longest common subsequence.

Solution: We need to maximize the number of matches; therefore, we can adopt the scoring scheme $\{1, 0, 0\}$. Mismatches and gaps do not contribute to the score. The maximum score is the maximum number of matches. The optimal alignment itself gives the set of matches, i.e. the longest common subsequence.

(c) Is the length of the longest common subsequence related to the edit distance in any way?

Solution: This example shows that it is not related in an intuitive way. Consider $x = AAAAAAAAAAB$ and $y = BCCCCCCCCC$. The longest common subsequence of x and y is B of length 1. The edit distance cannot benefit from this common subsequence. If this common subsequence were to be used, then transforming x into y will have to make 10 insertions (for As) and 10 dele-

tions (for C s) which leads to 20. However, by substituting every character we get only 10 operations and that's the edit distance. However, if we disallow substitutions, then the only way to transform x into y would be to use 10 insertions and 10 deletions. In this case, the

$$2LCS + \text{editdistance} = m + n$$

In general,

$$LCS \leq (m + n - \text{editdistance})/2$$

(d) We define a scatter of a string u of length n as a any string

$$V_0 u_1 V_1 u_2 V_3 \dots V_{n-1} V_{n-1} u_n V_n$$

where V_i are arbitrary strings of any lengths. Design an $O(mn)$ algorithm to find the shortest scatter of both x and y (*Hint*: relate this to the longest common subsequence of x and y).

Solution; First, a scatter of u has to have at least the length of u . We start with u being a scatter of itself. To make this a scatter of v as well, we need to add characters of v . To include the minimum number of additional characters, we identify a sequence of characters in u that is a sequence in v , i.e. the longest common subsequence. This can be found as in part (b). This takes $O(mn)$ time. Then we insert the rest of the characters of v in the right places and this takes an additional $O(n)$ time.

Problem 5: Circular DNA alignment

Consider two circular DNAs x and y of length m and n respectively.

We are after the optimal global alignment of x and y . This can be obtained as follows: Consider a circular shift of x , $x_i \dots x_m x_1 \dots x_{i-1}$ for some $1 \leq i \leq m$. Consider a circular shift of y , $y_j \dots y_n y_1 \dots y_{j-1}$ for some $1 \leq j \leq n$. Find their optimal global alignment, and repeat for every possible pair of circular shifts of x and y . Finally pick the highest scoring alignment.

Since there are m circular shifts of x and n circular shifts of y , the above algorithm will take $O(m^2 n^2)$ time.

(a) Design an $O(mn^2)$ time algorithm that will find the optimal global alignment of two circular DNAs x and y .

Solution: Consider an optimal circular alignment of x and y . If we cut the circular alignment at x_1 , we obtain an alignment of $x_1 \dots x_m$ with some circular shift of y that has the same score as the score of the optimal circular alignment. Therefore, to find the optimal circular alignment it is enough to compute the optimal scores for the alignments of $x_1 \dots x_m$ with all circular shifts of y and

return the alignment corresponding to the best score. So we compute n dynamic programming tables A_i , each of which takes $O(mn)$ time. Then we pick the highest score among the $A_i(m, n)$ in $O(n)$ time. The total time so far is $O(mn^2)$. We need an additional $O(m + n)$ to obtain the alignment itself, but this does not change the asymptotic bound of $O(mn^2)$.

(b) Can you obtain the optimal global alignments for all pairs of circular shifts (i.e. mn) of x and y in $O(mn^2 + nm^2)$?

Solution: We can build n tables A_1, \dots, A_n for aligning $x_1 \dots x_m$ with the n circular shifts of $y_1 \dots y_n$. We do the same for the reverse of x and y , i.e. we build n tables A'_1, \dots, A'_n for aligning $x_m \dots x_1$ with the n circular shifts of $y_n \dots y_1$. This so far takes $O(mn^2)$ time.

$A_k(i, j)$ is the optimal score of aligning $x_1 \dots x_i$ with the j^{th} prefix of the k^{th} circular shift of y . Similarly, $A'_k(i, j)$ is the optimal score of aligning $x_m \dots x_i$ with the j^{th} suffix of the k^{th} circular shift of y .

Now observe that an alignment of a circular shift of x , $x_i \dots x_m x_1 \dots x_{i-1}$ with a circular shift of y , $y_j \dots y_{j-1}$ corresponds to an alignment of $x_i \dots x_m$ with some suffix $y_j \dots y_k$ of the k^{th} circular shift of y followed by an alignment of $x_1 \dots x_{i-1}$ with some prefix $y_{k+1} \dots y_{j-1}$ of the $(k + 1)^{\text{st}}$ circular shift of y . We can try all the n possibilities for k and pick the best one, each of which takes $O(1)$ time given the information we pre-computed above.

Therefore, after the pre-computation step, the score of each of the mn alignments can be computed in $O(n)$ time, and each alignment can be obtain in $O(m + n)$ time (by obtaining the actual alignments for each of the two highest scores of the sub-alignments), resulting in $O(m + n)$ time for each of the mn alignments. The total running time of the algorithm is therefore: $O(mn^2) + mn \cdot O(m + n) = O(mn^2 + n^2m)$