```python
def align(
        seq1: str, seq2: str, match_award=-3, indel_penalty=5, sub_penalty=1, banded_width=-1,
        gap='-'
) -> tuple[float, str | None, str | None]:
    m, n = len(seq1), len(seq2)

    # Initialize the DP table with default value of float('inf')
    dp = {}

    # Initialize the first row and column
    dp[(0, 0)] = 0
    for i in range(1, m + 1):
        dp[(i, 0)] = i * indel_penalty
    for j in range(1, n + 1):
        dp[(0, j)] = j * indel_penalty

    # Fill in the DP table with either full or banded alignment
    if banded_width == -1:
        # Fill in the DP table with full alignment
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                match = dp.get((i - 1, j - 1), float('inf')) + (match_award if seq1[i - 1] ==
seq2[j - 1] else sub_penalty)
                delete = dp.get((i - 1, j), float('inf')) + indel_penalty
                insert = dp.get((i, j - 1), float('inf')) + indel_penalty
                dp[(i, j)] = min(match, insert, delete)
    else:
        # Fill in the DP table with banded alignment
        for i in range(1, m + 1):
            for j in range(max(1, i - banded_width), min(n + 1, i + banded_width + 1)):
                match = dp.get((i - 1, j - 1), float('inf')) + (match_award if seq1[i - 1] ==
seq2[j - 1] else sub_penalty)
                delete = dp.get((i - 1, j), float('inf')) + indel_penalty
                insert = dp.get((i, j - 1), float('inf')) + indel_penalty
                dp[(i, j)] = min(match, insert, delete)

    # Backtracking to reconstruct the optimal alignment
    alignment_seq1, alignment_seq2 = [], []
    i, j = m, n

    while i > 0 or j > 0:
        current_cost = dp.get((i, j), float('inf'))

        if i > 0 and j > 0 and (seq1[i - 1] == seq2[j - 1] or current_cost == dp.get((i - 1, j
- 1), float('inf')) + sub_penalty):
```

```
            alignment_seq1.append(seq1[i - 1])
            alignment_seq2.append(seq2[j - 1])
            i -= 1
            j -= 1
        elif i > 0 and current_cost == dp.get((i, j - 1), float('inf')) + indel_penalty:
            alignment_seq1.append(gap)
            alignment_seq2.append(seq2[j - 1])
            j -= 1
        else:
            alignment_seq1.append(seq1[i - 1])
            alignment_seq2.append(gap)
            i -= 1

    # Reverse the alignments to get the correct order
    alignment_seq1.reverse()
    alignment_seq2.reverse()

    alignment_cost = dp.get((m, n), float('inf'))

    return alignment_cost, ''.join(alignment_seq1), ''.join(alignment_seq2)
```

<u>**initialization of the DP**</u>

```
    dp {}

    for i in range(1, m + 1): # 0(m)
        dp[(i, 0)] = i * indel_penalty
    for j in range(1, n + 1): # 0(n)
        dp[(0, j)] = j * indel_penalty
```

The first row and column are initialized in 0(m +n) times where m = length of seq1 and n = length of seq2.
A dictionary is used to store the dp values. In the case of unrestricted alignment, the dp requires a space complexity of 0(m*n) since it's a 2d structure of size (m+1)*(n+1) and one entry is needed for each cell.

<u>**Filling the DP table**</u>

```
if banded_width == -1:
        for i in range(1, m + 1): # 0(m)
            for j in range(1, n + 1): # 0(n)
```

the unrestricted alignment (banded_width = -1), the loops iterate through all the cells of the table which leads to 0(m*n)
The rest of the line of code after "for j in range(1, n + 1):" take a constant time of 0(1)

The inner loop of the banded alignment is constrained by the banded width. So for each ‘i’, the ‘j’ index stretches for a limited range. If the banded width is k then the complexity of the alignment then becomes 0(m * k).
The band_width is used to constrain the matrix size. As the band for each row is being calculated the space complexity is 0(k*n)

```
for i in range(1, m + 1): # O(m)
            for j in range(max(1, i - banded_width), min(n + 1, i + banded_width
+ 1)): # O(k)
```

Backtracking

```
while i > 0 or j > 0:
```

the backtracking runs back the alignment by iterating through the sequence which takes 0(m + n) times in both cases.
The operations within the loop is 0(1)

Therefore, for **unrestricted alignment**:
Time complexity
Initialization: O(m + n)
Filling the Table: O(m * n)
Backtracking: O(m + n)

- Total time complexity: O(m * n)

Space Complexity
Space complexity: 0(m*n)

And for **banded alignment:**
Time complexity
Initialization: O(m + n)
Filling the Table: O(m * k)
Backtracking: O(m + n)

- Total time complexity: O(m * k)

Space complexity
Space complexity: 0(k*n)

Dependency Pointers

In the banded algorithm, the dependency pointers are modified to make sure that the cells within the band are accessible. The algorithm considers cells of (i-1, j-1) , (i-1, j), and (I, j-1) if they are within the band. This is to make sure the band doesn’t access cells outside of the band.