Student Name: Ashna Mittal

Student Number: 251206758

CS 3340B: Assignment 3

Solution 1:

To find the largest prefix of the pattern P that matches a substring of the text T, where the match must start with P1, we need to adjust the original KMP algorithm to focus on prefix lengths of P that can be matched against any part of T. The original KMP algorithm is designed to find occurrences of the entire pattern P within the text T and then terminate instantly. However, for this modification, the goal is to identify the longest prefix of P that matches consecutively within T, starting from any position in T, and the algorithm continues to execute until the end of T is reached.

Original KMP Algorithm Overview:

1) Preprocessing Phase: Compute the prefix function (π table) for P, which helps in determining how far back the search should jump in P upon a mismatch.
2) Matching Phase: Sequentially match P against T by using the π table to efficiently skip characters that have been matched previously, thus avoiding unnecessary comparisons.

Modified KMP Algorithm for Largest Prefix Matching:

The key modification to the KMP algorithm involves adjusting how mismatches are handled during the matching phase and potentially modifying the preprocessing phase to accommodate the requirement of finding the largest matching prefix rather than the entire pattern. Here's a conceptual approach to modifying the algorithm:

1) Preprocessing Phase: This phase remains largely unchanged. Compute the π table for P as usual, which will still be used to determine the next positions to consider after a mismatch.

2) Matching Phase:
   - Initialize two pointers, one for T (let's call it i) and one for P (let's call it q), at the beginning of their respective strings.
   - As you iterate through T, compare the current character in T(T[i]) with the current character in P(P[q + 1]).
   - If characters match, increment both i and q, and record q as the length of the current matching prefix.
   - If a mismatch occurs or P is exhausted (q = m), use the π table to find the longest prefix of P that matches up to this point and start the next comparison from this new position in P. The critical difference here is that upon a mismatch or successful match, you do not restart the matching process from P[1], but rather, you continue scanning T from the next character.
   - Keep track of the longest matching prefix length found during the scanning process. This is done by maintaining a variable that records the maximum value of q reached before each mismatch or successful prefix match.

3) Termination: The algorithm continues until the end of T is reached. The length of the longest matching prefix found is the maximum value recorded for q during the matching phase.

For Example: Suppose P = "abc"and T = "daabcdef". The modified algorithm would identify "abc" as the largest prefix of P that matches a substring of T, starting at position 4 in T.

Modified Algorithm Implementation:

- The π table computation remains as described in the course material. It's the matching phase that undergoes significant modifications to ensure that the search for the largest matching prefix is optimized and does not necessitate finding the entire pattern in T.
- Computing the prefix table takes O(m) time, where m is the length of pattern P. Scanning through string T takes O(n) time, where n is the length of T. At each step of scanning, the characters of P are compared with the corresponding characters of T, which are determined by the reference pointer for the string T. In the worst case, the prefix table will have to be consulted for each character comparison, and the total number of comparisons is limited by the length n of string T. Therefore, the time complexity of this algorithm is O(n + m). This

approach maintains the O(n + m) time complexity of the original KMP algorithm but shifts the focus from finding the entire pattern to finding the longest prefix match.

By following this new modified approach, the algorithm can efficiently find the largest prefix of P that matches a substring of T, adhering to the requirement that the match must start with P[1].

Pseudocode:

```
Algorithm LargestPrefixKMP(Text T, Pattern P):
    n = Length(T)
    m = Length(P)
    π = ComputePrefixFunction(P)
    q = 0  // Number of characters matched in pattern P
    longestMatch = 0  // Length of the longest matching prefix found

    for i = 1 to n do:
        while q > 0 and P[q + 1] ≠ T[i]:
            q = π[q]  // Using π table to find next position in P
            longestMatch = max(longestMatch, q)  // Updating the longest match found

        if P[q + 1] == T[i]:
            q = q + 1  // Increasing the number of matched characters

        if q == m:
            // The full pattern P has been matched
            q = π[q]  // Preparing for the next potential match in T

        longestMatch = max(longestMatch, q)  // Updating longest match after each character
match

    return longestMatch
```

Solution 2:

We will start sorting lexicographically the substrings of the given text "hippityhoppityboobob" in increasing lengths, starting from substrings of length 1 and doubling the substring length at each pass until we cover the entire string. This is done over log n passes where n is the length of the string.

- Step1: Initial Ranks – We will assign an initial rank to each character in the string based on its lexicographical order. For simplicity, let’s assume 'a' < 'b' < ... < 'z'.
- Step2: Sorting Substrings – We will then sort substrings of length 1 based on their ranks, then proceed to substrings of length 2, 4, etc., by comparing the ranks of their left and right halves.
- Step3: Updating Ranks - After each sorting step, we will update the ranks based on the new sorted order. Substrings that are identical will have the same rank.
- Step4: Repeat - Repeat the process, doubling the substring length each time until you've considered the entire length of the string.
- Step5: Suffix Array Construction - Once all substrings have been sorted and ranked, we will construct the suffix array by listing the starting indices of the suffixes in the order determined by the sorting process.

i	index	suffix
0	0	hippityhoppityboobob
1	20	b
2	18	bob
3	15	boobob
4	1	hippityhoppityboobob
5	8	hoppityboobob
6	2	ippityhoppityboobob
7	12	ityboobob
8	5	ityhoppityboobob
9	19	ob
10	17	obob
11	16	oobob
12	9	oppityboobob
13	11	pityboobob

14	4	pityhoppityboobob
15	10	ppityboobob
16	3	ppityhoppityboobob
17	13	tyboobob
18	6	tyhoppityboobob
19	14	yboobob
20	7	yhoppityboobob

Suffix Array: {20, 18, 15, 1, 8, 2, 12, 5, 19, 17, 16, 9, 11, 4, 10, 3, 13, 6, 14, 7}

Solution 3:

The Longest Common Subsequence (LCS) seeks to find the longest subsequence common to all sequences in a set of sequences (often just two). A subsequence is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements. To reconstruct an LCS from:

- The completed c table, which contains the lengths of the LCSs at each subproblem.
- The original sequences X (with elements x1, x2, ..., xm) and Y (with elements y1, y2, ..., yn).

This has to be done in O(m+n) linear time, and without using the b table that typically contains directional information used to reconstruct the LCS (which direction the value in the c table was derived from).

The pseudocode below is a function called COMPUTE -LCS that recursively follows the c table to print the LCS. It works as follows(assuming that the table c and sequences X and Y are 1-indexed, as is common in algorithmic pseudocode) :

- if c[i, j] == 0: The base case, if the table entry at c[i, j] is 0, there are no more characters to match, so it returns without doing anything.

- if X[i] == Y[j]: If the characters at position i in sequence X and position j in sequence Y match, the character belongs to the LCS. It recursively calls itself with indices i - 1 and j - 1 (moving diagonally up in the table), and then it prints the matching character.
- else if c[i - 1, j] > c[i, j - 1]: If the LCS length from the upper cell is greater than the length from the left cell, it recursively calls itself with indices i - 1 and j (moving up in the table).
- else: Otherwise, it calls itself with indices i and j - 1 (moving left in the table).

```
COMPUTE-LCS(c, X, Y, i, j)
  if c[i, j] == 0
    return
  if X[i] == Y[j]
    COMPUTE -LCS(c, X, Y, i - 1, j - 1)
    print X[i]
  else if c[i - 1, j] > c[i, j - 1]
    COMPUTE -LCS(c, X, Y, i - 1, j)
  else
    COMPUTE -LCS(c, X, Y, i, j - 1)
```

Solution 4:

We need to arrange a paragraph into lines where each word W_i has a length L_i. The width of each line on the screen is M characters. The objective is to minimize the total cost that penalizes the wasted space at the end of each line. The cost of a line with words from W_i to W_j is given by:

$$C(i,j) = (M - (j - i) - \Sigma_{k=i}^{j} L_k)^3$$

The cost is infinite (∞) if the sequence doesn't fit in a single line, and the cost is zero if it's the last line of the paragraph.

Solution using Dynamic Programming:

- Initialization:
    - Let dp[i] represent the minimum total cost for wrapping up to the word W_i.
    - Initialize dp[0] to 0 and dp[i] for i ≥ 1 to ∞.
- Bottom-up Calculation:
    - For each word W_i, calculate the minimum cost of wrapping from that word to the end by considering all possible end points for the lines following W_i.
    - Update the dp[i] array with the minimum total cost for each starting point.

- Cost Calculation:
    - For each pair pair of indices (i, j) , i ≤ j, calculate the the cost C(i,j) if the sequence of words from W_i to W_j can fit on one line; otherwise, set it to ∞.
    - If j = n (he last word), set C(i,j) to 0.
- Reconstruction:
    - After computing the dp array, reconstruct the solution by tracing back from dp[n] to determine the actual wrapping that achieves the minimum cost.

Pseudocode:

```
function ComputeWordWrap(W, L, M):

   n = W.length

   dp = array of size (n+1) with all values as ∞

   dp[0] = 0



   for i from 1 to n:

       for j from i to n:

           spaceLeft = M - (j - i) - sum(L[i...j])

           if spaceLeft >= 0:
            cost = (i == n) ? 0 : spaceLeft^3

            dp[j] = min(dp[j], dp[i-1] + cost)

        else:

            break  // No need to consider further j if this doesn't fit



    return dp[n]  // This contains the minimum total cost



function sum(L[i...j]):

    return sum of L[k] for k from i to j



// Call the function with the list of words, their lengths, and the maximum line width

minCost = ComputeWordWrap(W, L, M)
```

Time Analysis:

- The outer loop runs for n iterations (for each starting word).
- The inner loop also runs for n iterations in the worst case (for each possible ending word).
- The sum function may take up to O(n) time for each calculation.
- We can optimize the sum calculation by using a prefix sum array, reducing it to O(1) for each sum query.

With the optimization, the total running time of the algorithm is O(n^2) assuming constant time for sum queries with a prefix sum array.

Solution 5: Minimum spanning tree algorithms, such as Prim's or Kruskal's, traditionally prioritize edges with the smallest weights to ensure the spanning tree has the minimum possible total weight. Modifying a minimum spanning tree (MST) algorithm to find a maximum spanning tree (MaxST) in a graph involves prioritizing edges with the largest weights instead of the smallest edge at each step.

Kruskal's Algorithm for MaxST –
Kruskal's algorithm for finding a minimum spanning tree operates by sorting all the edges of the graph in non-decreasing order of their weights and then adding them one by one to the spanning tree, provided they don't form a cycle, until all vertices are connected.

Modification for MaxST:

- Sort the Edges Initially, sort all the edges in descending order by weight, instead of ascending order. (i.e., from largest to smallest).
- Select the Edges: Proceed as in the standard Kruskal's algorithm by adding the edges to the spanning tree if they do not form a cycle. Since edges are considered from largest to smallest, this process builds a maximum spanning tree.
- Cycle Check: Use the same method (e.g., Union-Find data structure) to check for cycles when adding a new edge to ensure that the resulting graph remains a tree.

Prim's Algorithm for MaxST –
Prim's algorithm grows the minimum spanning tree by starting from an arbitrary vertex and repeatedly adding the smallest edge that connects a vertex in the tree to a vertex outside the tree, until all vertices are included.

Modification for MaxST:

1) Priority Queue: In Prim's algorithm, a priority queue (or a similar structure) is typically used to select the next smallest edge that connects the tree to a new vertex. Modify this priority queue to prioritize edges with the largest weights instead.
2) Edge Selection: Continue using Prim's greedy approach, but in each step, choose the edge with the maximum weight that can be added to the growing spanning tree without forming a cycle.

3) Update Weights: Whenever a new vertex is added to the tree, update the weights of the edges connecting this vertex to the rest of the graph in the priority queue to ensure that the maximum weight edge is selected next.

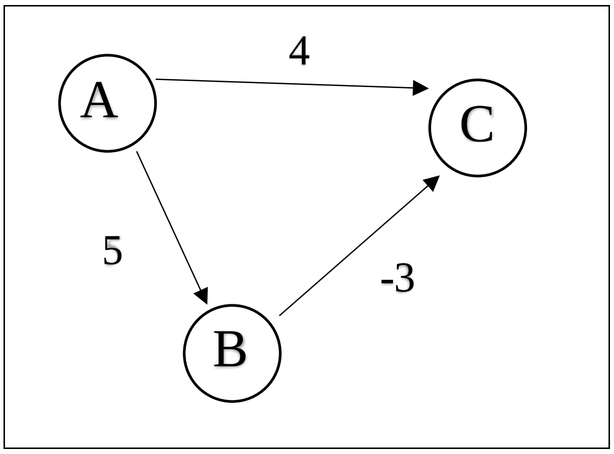
Both modified algorithms work well even if the graph contains negative weights, as the objective remains to maximize the total weight of the spanning tree. The time complexity of the modified algorithms remains the same as their minimum spanning tree counterparts. For Kruskal's, it's O(E log E) due to sorting, and for Prim's, it depends on the priority queue implementation (e.g., O(E + V log V) with a Fibonacci heap). Like MSTs, maximum spanning trees are not necessarily unique unless all edge weights are distinct.

While both Kruskal's and Prim's algorithms find minimum spanning trees, they can be adapted for maximum spanning trees. Kruskal's algorithm, using a greedy approach, sorts edges by weight, but it may not efficiently handle dense graphs. Prim's algorithm, on the other hand, starting from a vertex, grows the tree iteratively, making it more suitable for dense graphs.

Solution 6: To provide a counterexample demonstrating the failure of Dijkstra's algorithm with negative weight edges, let us construct a simple directed graph with three vertices A, B, and C, and incorporate negative edge weights. The essence of the counterexample lies in how Dijkstra's algorithm, which relies on the property that a shortest path from a source vertex to any other vertex in the graph will not get shorter by adding more edges, fails when negative weight edges are present.

Consider the directed graph G = (V, E) with vertices V = {A,B,C}and edge set E defined by the weights:

- w(A,B) = 2: the weight of the edge from A to B is 5.
- w(B,C) = −3: the weight of the edge from B to C is -3 (the negative weight edge).
- w(A,C) = 4: the weight of the edge from A to C is 4.



Dijkstra's Algorithm Process –

- Initialization: Dijkstra's algorithm initializes the distance to the source vertex A as 0 and all other distances to infinity. Thus, we have d(A) = 0, d(B) = ∞, d(C) = ∞.
- Relaxation from A: The algorithm begins at A, updating the distance to B to 5 (since 0+5 <∞) and the distance to C to 4 (since 0+4 < ∞). The updated distances are d(B)=5 and d(C)=4.
- Selection of Next Vertex: Suppose the algorithm selects the next vertex with the shortest tentative distance from A, which is B.
- Relaxation from B: The algorithm proceeds to relax the edge from B to C. The distance from A to C through B is calculated as 5−3= 2, which is less than the direct path from A to C. Hence, d(C) is updated to 2.
- Final Distances: The algorithm concludes with the shortest distances from A being d(A) = 0, d(B) = 5, d(C) = 2.

How the algorithm Failed –
Dijkstra's algorithm fails in the above example due to the presence of a negative weight edge, which leads to the re-evaluation and updating of the shortest path to a vertex (C in this case) after it has been determined. The algorithm assumes that once a vertex has been "visited" (i.e., its shortest distance from the source is finalized), its distance cannot decrease. However, the negative edge weight from B to C allows for a shorter path to C to be discovered after C's distance was initially set, violating this assumption.

Thus, the example graph demonstrates that Dijkstra's algorithm does not correctly handle graphs with negative weight edges, as it may fail to find the shortest path due to its greedy nature and the assumption that a path's length cannot decrease by exploring more vertices. In contrast, algorithms designed to handle negative weights, such as the Bellman-Ford algorithm, correctly adjust for shorter paths discovered as a result of negative weight edges.