CS3340B Winter 2021 Assignment 3 Due Apr. 6 2021
James Walsh jwalsh57 250481718

1. Modify the KMP string matching algorithm to find the largest prefix of P that matches a substring of T. In other words, you do not need to match all of P inside T; instead, you want to find the largest match (but it has to start with p1).

   Input: text T[1...n] and pattern P[1...m]
   Output: matching positions.

```
        KMP_prefix_matcher(T, P)
 1.  i:= 1;
 2.  q := 0;
 3.  max_q := 0;                        // introduce new variable to track longest prefix matched
 4.  max_positions = [];                // introduce new empty list to store positions of longest matched prefixes in T
 5.  while $i \le n$ do
 6.     if T[i]==P[q+1]
 7.        $i := i + 1;$
 8.        $q := q + 1;$
 9.        if (max_q == q)              // found a partial match of greatest length (so far)
10.           max_positions.append(i-q);  // append current position in T to the list of positions
11.        else if (max_q < q)          // new longest prefix found
12.           max_q = q;                // update length of longest prefix found
13.           max_positions.clear();    // clear list of positions (contents were from shorter matches)
14.           max_positions.append(i-q);  // append current position in T to the list of positions
15.     else
16.         if q==0 then
17.           $i := i + 1;$
18.        else
19.           $q := next[q];$
20.     if q==m then
21.        print "Full pattern found at position" i-m;
22.        q = next[q];
23. if (0 < max_q && max_q < m)         // only a partial match was found
24.     print("partial match of length " i " found at positions ");
25.     print(max_positions);           // print elements (positions in T) in list
```

Description: see comments in pseudocode
Correctness: This modification to the KMP string matching algorithm adds new variables to track the length of the longest prefix matched and the locations where they were matched. After the entire text has been traversed, that information is then printed as required.
Time Complexity: The modification just adds some constant work before and after the loop. It also adds some constant work to the step when a character is matched (lines 9-14) but it does not alter the number of steps, so the time complexity is unchanged from the original KMP String_Matching algorithm's complexity of ***O*(*n*)**.
Space Complexity: The position list could be n elements long in the worst case (the pattern is 1 character and matches every character in T) so the space complexity is now **O(n)** (this is larger than the original space complexity of storing the next table which was O(m).

2. In the textbook, 15.4-2 (pp. 396).

```
reconstruct_LCS(X, Y)
1.    i := X.length
2.    j := Y.length
3.    reconstruct_LCS_recur(X, Y, i, j)

reconstruct_LCS_recur(X, Y, i, j)
1.    if (i==0 or j==0)
2.        return
3.    if ($X_i$==$Y_j$)
4.    reconstruct_LCS_recur(X, Y, i-1, j-1)
5.    print($X_i$)
6.    else if (c[i, j-1] <= c[i-1. j])          // move up 1 row (includes case where c[i, j-1] == c[i-1. j] and either can be chosen)
7.    reconstruct_LCS_recur(X, Y, i-1, j);
8.    else                                      // move left 1 column
9.    reconstruct_LCS_recur(X, Y, i, j-1);
```

Description: This reconstruction functions by checking whether the characters in X and Y at the current location in the c table match. If they do, the algorithm moves diagonally to the upper left. If not it moves to whichever of the 2 possible subproblems has the greater value.
Correctness: For any location c[i, j], the next location can be determined using only the c table by comparing Xi and Yj (if they match, move up and left), and if needed compare the values of c[i-1, j] and c[i, j-1] (and moving to the larger one).
Time Complexity: At least 1 of i or j is decremented every recursive call so worst case complexity is **O(m+n)**
Space Complexity: Nothing new needs to be stored so the complexity of the c table still dominates **O(m*n),** even though the O(m*n) space of the b table has been saved**.**

3. In the textbook, 16.2-5 (pp. 428). real line points

Description: (assume that the points are sorted in the input) Start with the leftmost point, draw an interval of length 1 to the right. Repeat from the smallest “uncovered” point until all points are covered.
Correctness: Start by deciding where to place the leftmost interval. It obviously needs to include the leftmost point. There is no reason to cover any area to the left of the leftmost point so start the leftmost interval at the leftmost point and have it cover 1 to the right (where it may or may not cover other points as well). The problem is now reduced. Follow a similar reasoning for deciding where to place the next (2nd leftmost) interval at the new leftmost “uncovered point”. Continue until all points are covered.
Time Complexity: *O*(n) In the worst case, no points are within 1 of each other so an interval/line segment needs to be created n times for each point. (If the points are not sorted in the input, then the complexity of the sorting algorithm will likely dominate)
Space Complexity: O(1) Intervals do not need to be stored and can be printed/used immediately.

4. In the textbook, 16.3-8 (pp. 436). Huffman coding proof

Let C be the character set {c_1, c_2, … c_{255}. c_{256}} with $c_1.freq \geq c_2.freq \geq ... \geq c_{255}.freq \geq c_{256}.freq$ and $c_1.freq < 2c_{256}.freq$
The first 2 nodes merged are c_{255} and c_{256} creating an internal node with a frequency greater than c1.freq. The next 2 nodes merged are c_{254} and c_{253} creating an internal node with a frequency greater than or equal to any pre-existing node. This continues until the entire row of internal nodes has been created. This new row has the same properties as the original row (max frequency < 2 min frequency). And the process repeats until a full perfect binary tree with 256 leaves and height 8 is created. Thus, Huffman coding produces an 8-bit code for each character that is encoded, which is the same efficiency as using standard 8-bit encoding.
QED

5. Solve the following variation of the 0-1 knapsack problem (pp. 425): The assumptions are identical to those of the 0-1 knapsack problem, except that there is an unlimited supply of each item. In other words, the problem is to pack of maximum value with items of given weights and values in a knapsack with a given-weight, but each item may appear many times.

$$c[i,w] = \begin{cases} 0, & \text{if } i=0 \text{ or } w=0 \\ c[i-1, w] & \text{if } w_i > w \\ \max(\mathbf{v_i+c[i, w-w_i]}, c[i-1, w]) & \text{else.} \end{cases}$$

Description: This solution modifies the solution to the 0-1 knapsack problem by allowing the algorithm to choose a value from the same row of the table (which allows for the possibility of putting several of the same item in the knapsack) if sufficient weight is available.
Backtracking now has to perform an additional check to see which case applies, and in the event of the new case print(i) and $w \leftarrow w - w_i$ (i stays the same).
Correctness: From the optimal solution which includes item j, if item j is removed then the remaining load must be the most valuable load possible that uses weight $W-w_j$.
Time Complexity: Still only need to fill the same table (O(nW)). Backtracking is now O(n+W) (previously it was O(n)) because now there is also the possibility of moving left through the columns and staying on the same row at each step, whereas before there was a guarantee to move up a row at every backtracking step. Overall time complexity is still **O(nW)**.
Space Complexity: O(nW). Still only need to maintain the same table.

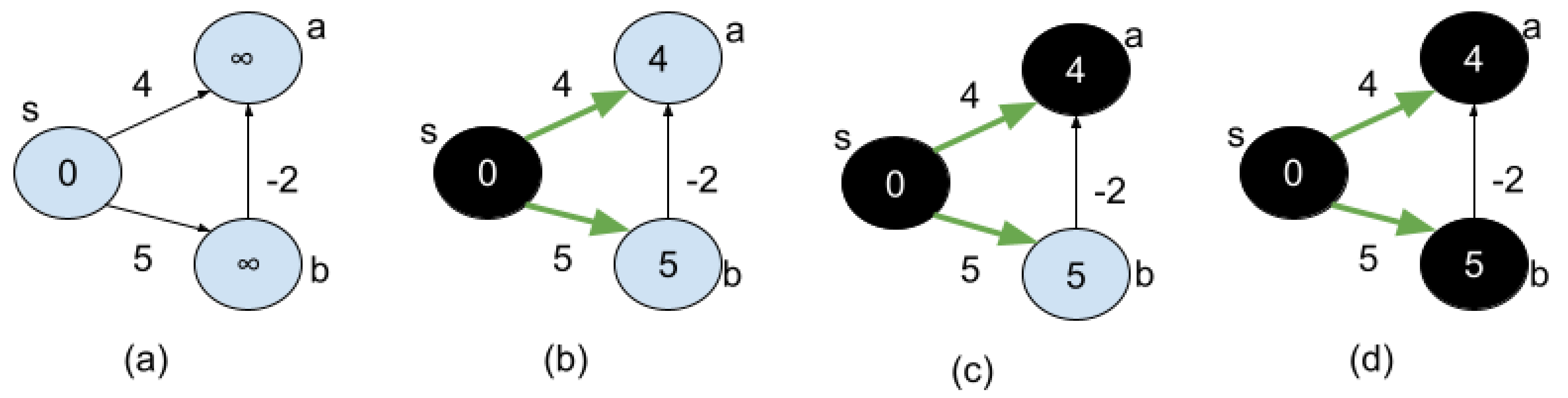
6. Modify minimum spanning tree algorithm to find maximum spanning tree.

```
MaxSpanningTree_Kruskal()
  1. put all edges in a max heap, put each vertex in a set by itself
  2. while not found a Max Spanning Tree yet do begin
  3.         delete max edge, {u, v} from the heap;
  4.         if u and v are not in the same set
  5.                 mark {u, v} as tree edge;
  6.                 union sets containing u and v;
  7.         if u and v are in the same set
  8.                 do nothing;
  9. end
```

Description: Modifies Kruskal's algorithm to produce a maximum spanning tree by extracting the maximum weighted edges instead of the minimum.
Correctness: Consider a tree T, and a tree Tneg which is identical to T except the weight of every edge has been multiplied by -1. The minST(minimum spanning tree) version of Kruskal's algorithm would produce a minST for Tneg. This minST of Tneg would be structurally identical (be composed of the same edges) to a maxST of T (which is what would be produced by this modified maxST version because max(a, b) yields the same choice as min(-a, -b)). Since the minST version of Kruskal's algorithm is correct, so is this modified maxST version.
Time Complexity: same as Minimum Spanning Tree
$O((|V|+|E|)\log(|V|))$ for heap operation
$O(|E|\log^*|V|)$ for union-find operation
Total $O((|V|+|E|)\log(|V|))$ time.
Space Complexity: Heap O(|V|), union-find O(|V|), **Total O(|V|)**

7. Find a counter example with three vertices that shows Dijkstra's algorithm does not work when there is negative weight edge.

Thick green arrows show the current shortest paths from s



The value of the shortest path to node a is locked in after step c. The arrow from b to a is then ignored in step d. So the final value at a is 4 from the S->a path instead of 3 from the S->b->a path.

8. Let G = (V, E) be a weighted directed graph with no negative cycle. Design an algorithm to find a cycle in G with minimum weight. The algorithm should run in time O(|V|^3).

```
find_Minimum_Cycle(G)
  1.  Use All-Pair Shortest-Paths (Floyd-Warshall or repeated |V| Dijsktra's calls) // computes d[u,v] for all pairs of u,v ∈ $G$  // O(|V|$^3$) or O(|V|(|V|+|E|)log(|V|)) time
  2.  min_cycle_weight = ∞
  3.  for each pair u,v where $u \neq v$        // O(|V|$^2$) pairs
  4.      w = d[u, v] + d[v, u];               // compute weight of cycle
  5.      if (w < min_cycle_weight)            // check if better than current best
  6.          min_cycle_weight = w;            // update weight of minimum cycle
  7.          v1 = u;                          // update pair of vertices that define the cycle
  8.          v2 = v;
  9.  if(min_cycle_weight == ∞)
  10.     print("No cycles found.");
  11.     return -1;
  12. path = [ ]                               // declare empty array/list to contain sequence of vertices in min weight cycle path
  13. backtrace(path, v1, v2);                 // trace path from v2 back to v1
  14. path.pop();                              // pops v1 off the end of the path so that we don't end up with a duplicate from the last element of the 1st call, and the 1st element of the 2nd call
  15. backtrace(path, v2, v1);                 // trace path from v1 back to v2
  16. path.reverse()                           // reverse the elements in the backtraced path so that it goes the correct direction along the directed edges.
  17. return (path, min_cycle_weight);
```

```
backtrace(path, i, j)
  1.  while $j \neq i$           // loop through vertices from j back to i
  2.      path.append($j$)       // append current vertex to the path
  3.      $j = \pi[i, j]$        // j becomes previous vertex on the path
  4.  path.append($i$)// loop doesn't cover i so have to append it after loop finishes
  5.  return path;
```

Description: This algorithm starts by solving the all-pair shortest paths problem using the Floyd-Warshall algorithm (alternately |V| applications of Djikstra's algorithm can be used, if G is sparse and has no negative edges, for better time complexity) to populate the D and Π tables. All cycles in G are checked and the minimum weight cycle is chosen. The 2 vertices that define the cycle are then used to trace its path. The path is then returned along with the weight.

Correctness: In a directed graph, one way to define a cycle is to define a path from a source vertex to a destination vertex (which is not the source vertex) and then back to the source vertex. By looping through the entire D table computed by the Floyd-Warshall algorithm, all such cycles can be checked and their weights can be computed. Once the minimum weight cycle has been identified, the 2 vertices that define it can then be used to retrace the path of the cycle. The path returned is from v2 to v1 and back to v2. Since it's a cycle, the order of the vertices is arbitrary, so this suffices to define it.

Time Complexity: Calling Floyd-Warshall in line 1 runs in O(|V|^3) time. The loop in line 3 runs O|V|^2 times (the work in the loop body is O(1)) so lines 3-8 run in O(|V|^2) time. The 2 backtrace() calls each run in O(|V|) time. The dominant work is the Floyd-Warshall call in line 1 and the total time complexity is **O(|V|^3)** (unless the Floyd-Warshall call can be replaced with |V| Dijkstra calls, in which case the total complexity is O((|V|+|E|)log(|V|) + |V|^2)).

Space Complexity: O(|V|^2) for the D andΠtables, O(|V|) for the path of the cycle. Total Space complexity is **O(|V|^2)**.