

# CS 3340 Assignment 3

Matvey Skripchenko 250899673

Due: April 6th, 2021

**Question 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  $p_1$ ).

*Solution:*

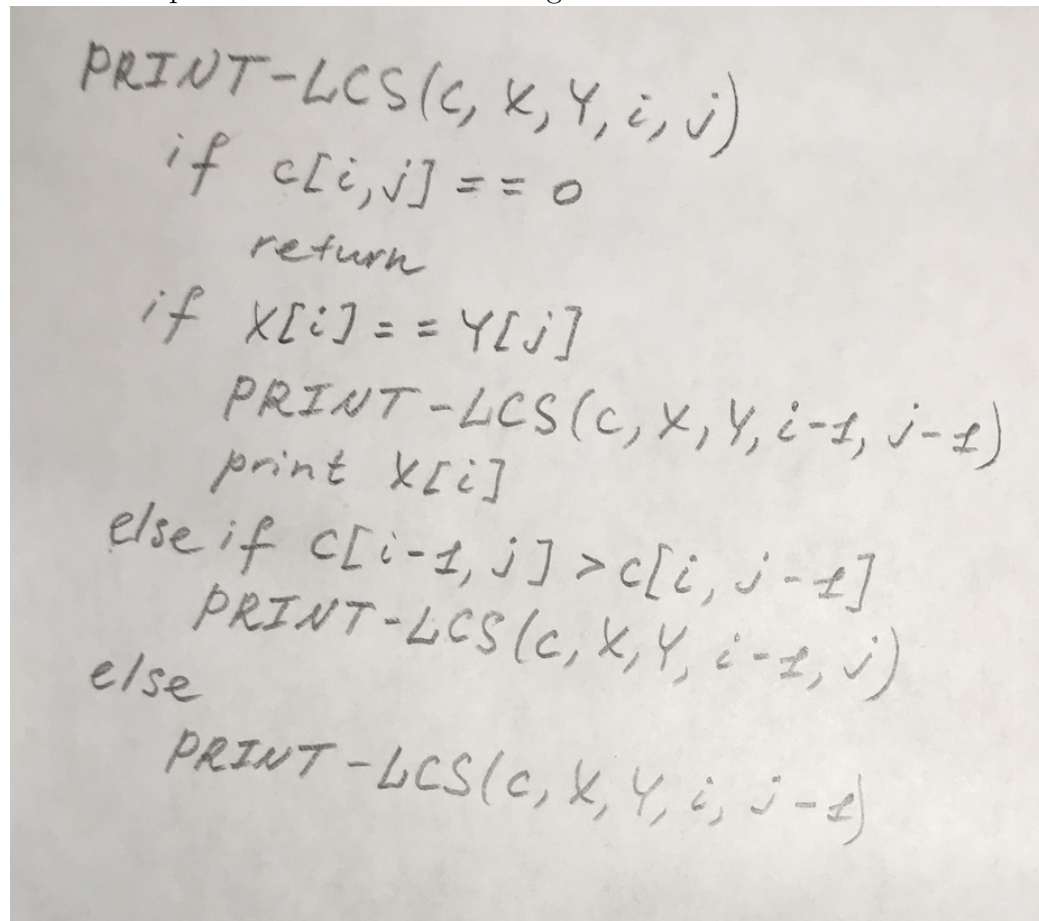
We know that when KMP is scanning the text, the state of KMP shows the length of the largest prefix of the pattern that matches that text up to the current point. Therefore, the maximum length up to that point can be recorded as well as the point/index in the pattern at which it was seen. Then, this recorded information can be used to find the longest matching prefix of P that at this stage is obviously a substring of T. Essentially we know that variable  $q$  takes care of the number of characters matched, so at some point, the max of  $q$  will give us the size of the longest prefix. But, since that variable always changes, we then need a variable, to keep track of the maximum of  $q$  as well as the variable to keep track of the index where longest matches occur,  $L_{max}$  and  $i_{max}$ , respectively. We can initialize these variables to zero before the for-loop. Variable  $q$  will function the same way by being incremented every time there is a match and then, we can have an if-statement comparing whether the value of  $q$  is greater than the value of  $L_{max}$  and thus, if greater, then  $L_{max}$  takes the value of  $q$ , and  $i_{max}$  will take the value of  $i - q + 1$ , what will keep the track of the index where the longest matches occur.

Thus, every time  $q$  is larger than  $L_{max}$ , we assign the value of  $q$  to that variable and  $i_{max}$  will take the value of  $i - q + 1$ . Therefore, eventually  $L_{max}$  will hold the longest matched sequence and thus, along with  $i_{max}$  index of the longest match, be used to obtain/print the longest prefix P that is a substring of T. So, at the end of the program execution, if no other matches are found, the variables  $L_{max}$  and  $i_{max}$  will contain the correct information for outputting the longest prefix match found.

Since we have only added an if-statement that will produce 2 operations for every match, the time complexity will remain the same as the original KMP, which is  $O(n + k)$ .

**Question 2.** Give pseudocode to reconstruct an LCS from the completed  $c$  table and the original sequences  $X = \langle x_1, x_2, \dots, x_m \rangle$  and  $Y = \langle y_1, y_2, \dots, y_n \rangle$  in  $O(m + n)$  time, without using the  $b$  table.

Here is the pseudocode for the asked algorithm:



```

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

Essentially, the entry in every array location  $c[i, j]$  depends on the other  $c$  table entries, i.e. all other  $c[\dots]$  mentioned in the pseudocode that are not  $c[i, j]$ . The way algorithm operates is through the recursively called function PRINT-LCS for each value  $i$  and  $j$ . So, we have the conditional statement executed for a minimum of  $m$  times for the index  $i$ , and for a minimum of  $n$  times for array index  $j$ . Thus, putting everything together, we have the total time complexity of  $O(m + n)$ .

**Question 3.** Describe an efficient algorithm that, given a set  $\{x_1, x_2, \dots, x_n\}$  of points on the real line, determines the smallest set of unit-length closed intervals that contains all of the given points. Argue that your algorithm is correct.

Essentially, this algorithm will work in a way where at each stage, the interval will be found as well as the new set (optimal) to be solved (should be smaller than previous set) that will consist of the elements not yet covered in the interval.

Firstly, the algorithm will sort the set of points given on the real line  $X = \{x_1, x_2, \dots, x_n\}$ . Secondly, the first interval obtained will be  $[x_1, x_1+1]$ , which will be added to a set  $UI$  that is empty to start off. Thirdly, all the points that are not covered in  $[x_1, x_1+1]$  are removed. Fourthly, the first and second step must be repeated until the given set will become empty. Lastly, the set  $UI$  is returned, which will be the small set having unit-length intervals that covers all the points in the set  $X$ .

If we visualize the leftmost interval, it must cover the leftmost point, but this interval will be irrelevant if it gets extended further than the leftmost point. Thus, knowing that interval's left hand side is exactly the leftmost point, then any point within a unit distance of the leftmost point is removed since otherwise they would be covered in this single interval. So, as mentioned before, we repeat until all the points are covered. Now, we can see that at every step there is a clear optimal choice for where the leftmost interval is assigned, meaning that the final solution would, in fact, also be optimal. This is because the final solution is the union of all intervals, so the notion of optimality extends throughout the whole solution.

**Question 4.** Suppose that a data file contains a sequence of 8-bit characters such that all 256 characters are about equally common: the maximum character frequency is less than twice the minimum character frequency. Prove that Huffman coding in this case is no more efficient than using an ordinary 8-bit fixed-length code.

*Proof:*

If we consider file where each character is of 8-bit fixed length, then for the file containing  $n$  characters,  $8n$  bits are needed to encode all the characters. We should take into account that Huffman algorithm produces a full binary tree, where a prefix is assigned to every character. So, let us consider the data file with  $m$  different characters,  $m = 2^j$ . Then we can look at the maximum and minimum character frequencies,  $F_{maximum}$  and  $F_{minimum}$ , respectively. In order for Huffman algorithm to produce a full binary tree with all the leaves at depth  $j$ , the data file must have characters such that  $F_{maximum} < 2 * F_{minimum}$ . Because now every character will have depth  $j$ , then every would need  $j$  bits in order to be encoded. We are given data file having 256 different characters. Thus,  $m = 256 = 2^8$ , so  $j$  is 8 and characters in the file are such that  $F_{maximum} < 2 * F_{minimum}$ . Hence, the result of the Huffman algorithm will be a full binary tree and thus all 256 characters will have a depth of 8. Due to every character having a depth of 8, the number of bits needed to encode all characters in the file, i.e. the cost, is  $8n$ . We can see that Huffman coding is therefore no more efficient than ordinary 8-bit fixed-length code if the data file consists of characters such that  $F_{maximum} < 2 * F_{minimum}$ .

**Question 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.

I think that there are actually multiple ways of solving this, i.e. multiple ways of filling the knapsack. Since there is an unlimited supply of each item, then 1 or more instance of any item can be used. Because items are unlimited and repetitions are allowed, we can then use a 1-dimensional array, `max_cap[W_int + 1]` such that `max_cap[i]` will store the max value that can reached using all items and the capacity  $i$  of the knapsack. Here is the snapshot of my python code:

```
def knapsack(W_int, n_vals, vals, weights):
    # max_cap[i] will store maximum
    # value with knapsack capacity i.
    max_cap = [0 for i in range(W_int + 1)]

    # Will fill up max_cap[] using recursion
    for i in range(W_int + 1):
        for j in range(n_vals):
            if weights[j] <= i:
                max_cap[i] = max(max_cap[i], max_cap[i - weights[j]] + vals[j])

    return max_cap[W_int]

# Test
W_int = 12
vals = [10, 20, 30, 70, 90]
weights = [1, 3, 5, 7, 9]
n_vals = len(vals)

print(knapsack(W_int, n_vals, vals, weights))
```

For testing I have used 5 items that have values: 10, 20, 30, 70, 90 with weights: 1, 3, 5, 7, 9, respectively for each item. And finally, I used  $W$  to be 12. The output of the program was 120, which is the maximum value given the stated inputs. Thus, in this case, the thief can get max value by taking 12 items of weight 1 having value of 10.

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

We could change the definition of T in MST to be the spanning tree having maximum possible weight and the order of data structure that contains the nodes of G. So, if we apply this to Prim's algorithm make all the keys initialized to -1, r to be zero and Q be a max-priority queue instead of the min-priority queue. Thus, we want the first node in the queue to be the one with the max weight. So, by going through the algorithm, we will loop until Q will be empty and achieving this by extracting the highest weight from Q. Hence, by using the value of the max node and not forgetting to check the adjacent nodes, we check if the node v of the current edge (u, v) is in Q and whether the edge's weight is larger than v's key.

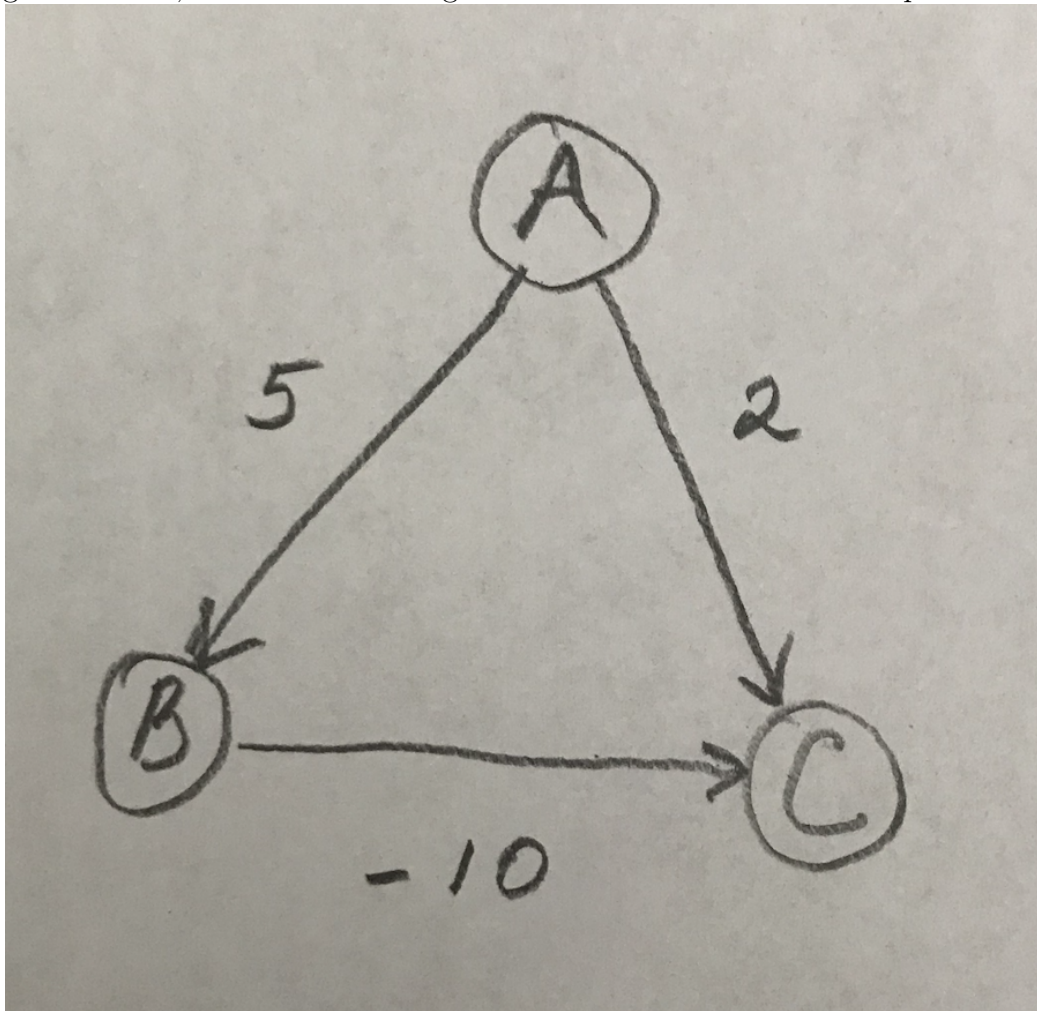
The described algorithm is correct since we first start with the initial node A at the root because it will have the highest weight in Q. So, after every iteration, the key of the adjacent node currently in Q is changed to be the weight of the connected edge in order for the maximum node to be extracted on the next go. Due to every node being extracted will have the highest value, which is determined by the highest weighted edge, the maximum spanning tree is obtained by continuously connecting the nodes through the most expensive edge.

The time complexity for this algorithm depends on the data structure chosen to implement Q and since we use max-heap, the run time of the algorithm will be  $O((|V| + |E|)\log|V|)$ .

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

*Solution:*

We know from Dijkstra's algorithm that when the vertex is marked as being closed, then the algorithm essentially thinks that the shortest path to that node was found. Therefore, the algorithm is done with this node and it will not be developed again due to the algorithm's assumption of the produced path being the shortest. However, if we throw in some negative weights in there, then the above might not be true. Here is the example:





Here, we will have  $V = \{A, B, C\}$  and  $E = \{(A, C, 2), (A, B, 5), (B, C, -10)\}$ . The algorithm going from A will first develop C, but will later on fail to find  $A \rightarrow B \rightarrow C$ . The reason for that is the algorithm will close the node A with value of zero. Then, the minimum valued node is being looked for, the minimal node is C, since B is 5 and C is 2. Thus, the algorithm will close C with the value of 2 and not look back, and later when B will be closed, then the value of C cannot be modified since it has already been marked as closed. Hence, in order to solve this issue, we are missing the "look back" tool which is achieved using Bellman-Ford that is able to look back in a recursive fashion, thus being able to cope with negative weights.

**Question 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  $O(|V|^3)$ .

*Solution:*

We know that Floyd-Marshall's algorithm will produce the minimum path between all the node pairs. Thus, if we run this algorithm on the given abstract graph and go through all the node pairs trying to find the cycle and not forgetting to record the cycle with the lowest weight, then the minimum is cycle is successfully obtained.

Assume we have already run Floyd-Marshall's algorithm on the graph and at the finish line, we will have the arrays containing shortest paths between the pairs. Then, we would want to create variables that keep track of the resulting pair and its respective weight. Then, by looping through every vertex u and v, we are checking for existence of the paths (u,v) and (v,u). If there is such a path, then the cycle has been found and we would want to compare its weight with the currently minimal weight. That means if  $w(u, v) + w(v, u) < \text{minimum}(\text{currently})$ , then the current minimum will be replaced and the node pair to u and v is updated, accordingly.

The algorithm is correct since we have the slight modification to Floyd-Marshall's algorithm. Thus, if we assume that Floyd-Marshall's algorithm returns the correct shortest paths for the pairs and there is an absence of negative cycles, then our modification being the loop that checks for cycles and minimum weights will make sense in terms of correctness. After performing the loop for every vertex pair and checking for a potential cycle for every possible node pair, we obtain the cycles. Then for each cycle obtained,

it is then compared with the current minimum. Thus, even in the case where we get a smaller cycle within a bigger one, then the smaller cycle is still returned.

In order to analyze the time complexity of such algorithm, we start of with the runtime for Floyd-Marshall's algorithm  $+ |V| * |V| *$  (two conditional statements  $+ 2$  operations at const time). This gives  $O(|V|^3) + O(|V|^2) * O(1)$  and thus, only leaving the dominating term we have  $O(|V|^3)$  time complexity.