Ben Dumais 250669195 3340-Asn 3

Question 1

Using the same logic and setup for an LCS as the textbook outlines, the PrintLCS can be rewritten using only c[i,j]. By checking the position directly above and to the left for an equal value before traversing diagonally, we can follow the same path through c.

Start by checking the position directly above the current one. If it has an equal c value, call PrintLCS recursively on that position and then print x_i after returning. If not equal, check the space directly to the left: if equal, call PrintLCS on that position then print x_i. If both the above are not equal, then the spot diagonally up and to the left (ie i-1,j-1) is one less, so call PrintLCS on that position but do not print.

Pseudocode: (as the question in textbook asks for it)

PrintLCS(c,X,i,j)

```
If i==0 or j ==0 return
if c[i,j] = c[i-1,j]
        PrintLCS (c,X,i-1,j)
        print x_i
else if c[i,j] = c[i,j-1]
        PrintLCS(c, X, I, j-1)
        print x_i
else
        PrintLCS(c, X, i-1, j-1)
```

Correctness: following the output example in textbook, fig 15.8, the algorithm retraces the correct path and prints correct characters from X.

Path, starting from i=m and j=n: c[7,6], c[6,6] + print, c[5,5], … , c[2,1] + print, c[1,0]

Complexity: each call of PrintLCS will either run in constant time when i or j is 0, or decrement i or j before calling itself again. Each call performs constant time (a return, or a print). In worst case scenario, we would iterate either all the way to the left then up, or all the way up then left (rather than diagonally which would decrement both and i and j). Since each call runs in constant time, and we call the function m then n times, the final complexity is m*O(1) + n*O(1) = O(m + n)

Question 2

If the order of increasing weight is the same ordering of items by decreasing value, then the highest valued items are also the lowest weight.

A	B	C	D	E	F	G	H	I	J
Lowest weight									Highest weight
Highest value									Lowest value

From this, it is obvious that the optimal item to take is whichever is furthest to the left (ie, highest value + lowest weight), as it adds the most value for minimal weight. A simple solution then can be made by looping through the items.

Algorithm: Assume the items are pre-sorted and in an array passed, A, in along with the max weight, W, and numbers of items, n. Loop through array from first (lowest weight/highest value) to last (highest weight/lowest value) while the sum of weights, w <= W. In each iteration, if weight of current item, wi, plus current w is less than or equal to W, add that item. Once we cannot add an item, break from loop and return. We break from algorithm since if we cannot add item k due to weight restrictions and since items k+1…n weigh more than k, we cannot add any more items.

Correctness: This algorithm only works when the order of items by increasing weight is the same as the order of decreasing value. Because of this, each item is both more valuable and lighter than any item that comes after it. It follows from this that each time we take the lightest item we are taking the most valuable and optimizing both remaining space and value at the same time.

Complexity: Since the array is passed in pre-sorted, the algorithm, in worst case scenario, iterates through the entire array, performing constant operations each pass (a conditional and then incrementing v and w). As such, complexity is given as n*O(1) = O(n) for worst case.

Question 3

Algorithm: Assuming that the points {x1,…,xn} are sorted (if not, just sort them prior), we need to loop through the points, create some set interval from the first point, then remove all points contained within this interval from the list before continuing to loop. If we use a set length of l, we delete all points x1 to xk, where k is the right-most point in the interval x1 + l. Upon completion, we will have a number of intervals, length l, that have covered all the points in the set. We return this number as the minimal number of intervals needed given length l, which can be passed into the algorithm.

Correctness: since l is fixed, if we take the furthest left point in the set and create an interval starting from that point, that is the maximum amount of points we can possibly cover from the start. Since we are removing any points between that point and (point + l) from the list, we ensure that we have taken out the maximum number of points possible starting from the original point before selecting the next one.

Complexity: In worst case, no other points are contained the intervals we set besides the one extracted from the list. Assuming we don't sort the list, we loop through maximum n points and perform constant operations for each, giving us O(n).

Question 4

Number of characters in a tree is given as $n = 2^k$ -> for n = 256, k = 8. To show that the size of a Huffman tree is the same as a fixed-length encoding of 8 bits when frequencies are the same, we must show that when length of code is equal for all characters the resultant tree is the same size as regular encoding.

Hypothesis

If we have 2^k characters, then every character has code length k

Base case

For k = 1, we have two leaf nodes connected by a root. K, the length of Huffman code, is also equal to 1

Inductive Step

Since leaf nodes will be chosen first when building a Huffman tree over composite nodes, at n/2 merges we have no leaf nodes left. This is especially true as all leafs have relatively the same frequency, so the frequency of a composite node is about 2f.

After n/2 merges we have n/2 composite nodes, or 2(k-1). If we replace the nodes with characters, the maximum frequency is still less than twice the minimum frequency (max frequency is less than twice that of lowest, from question)

By hypothesis, the new characters will have length k-1. If we replace the characters with the proper nodes, we now get the code tree for each character with length k. All leafs will be on the same level, thus have the same character length.

So in this case, if we have $n = 256 = 2^k = 2^8$. If 8 is the length of Huffman code and each leaf is on the same level, then we are left with a tree of the original size. Therefore, when frequencies are relatively the same, Huffman encoding is no longer efficient.

Question 5

Using input string P = babbabbabbababbabb and the next[] function from textbook:

m = 18

Q value	Next[]
1 – Pre Loop	[0]
2	[0,0]
3	[0,0,1]
4	[0,0,1,1]
5	[0,0,1,1,0]
6	[0,0,1,1,0,1]
7	[0,0,1,1,0,1,1]
8	[0,0,1,1,0,1,1,0]
9	[0,0,1,1,0,1,1,0,1]
10	[0,0,1,1,0,1,1,0,1,1]
11	[0,0,1,1,0,1,1,0,1,1,0]
12	[0,0,1,1,0,1,1,0,1,1,0,1]

13	[0,0,1,1,0,1,1,0,1,1,0,1,2]
14	[0,0,1,1,0,1,1,0,1,1,0,1,2,3]
15	[0,0,1,1,0,1,1,0,1,1,0,1,2,3,4]
16	[0,0,1,1,0,1,1,0,1,1,0,1,2,3,4,5]
17	[0,0,1,1,0,1,1,0,1,1,0,1,2,3,4,5,6]
18	[0,0,1,1,0,1,1,0,1,1,0,1,2,3,4,5,6,7]

Final values: q=18, k = 7, next = [0,0,1,1,0,1,1,0,1,1,0,1,2,3,4,5,6,7]

Question 6

Algorithm: Longest Prefix can be found by maintaining same code for string matching but adding in two additional variables and a conditional operation. q_{max} and i_{max} are the value of longest q matched and index of longest match occurring.

Before entering the for loop, initialize q_{max} and i_{max} to 0. Whenever a match occurs (ie, we increment q), check if the current value of q is greater than q_{max}. If greater, update q_{max} to the value of the current q, and update i_{max} to i-q+1.

Correctness: if we have a q_{max} of 5, i_{max} of 10, and currently q=6 and i=20, we have found another, longer match further down the target string. With the new operations, we now update q_{max} to 6 and i_{max} to 20-6+1 = 15. This makes sense, as we have matches 6 characters as if position 20, meaning T[15…20] are matches. We then continue to follow the algorithm until we finish looping. Assuming we do not find another match, we now print q_{max} characters starting from position i_{max} in T, giving us the longest prefix.

Complexity: The changed algorithm runs in the same time as the original KMP, but with an added conditional statement and 2 operations per match. The complexity remains O(n + k)

Question 7

Algorithm: The MST algorithm works by finding the lightest edge connecting the current MST and nodes outside of the current tree. To help achieve this, MST algorithms use a data structure to store nodes not currently in the tree in a way that allows us to easily extract the minimum key value. This method ensures we are always considering the lightest edge first, and by comparing if node v of the current edge (u,v) is currently in the MST (A) we can build a full MST (T).

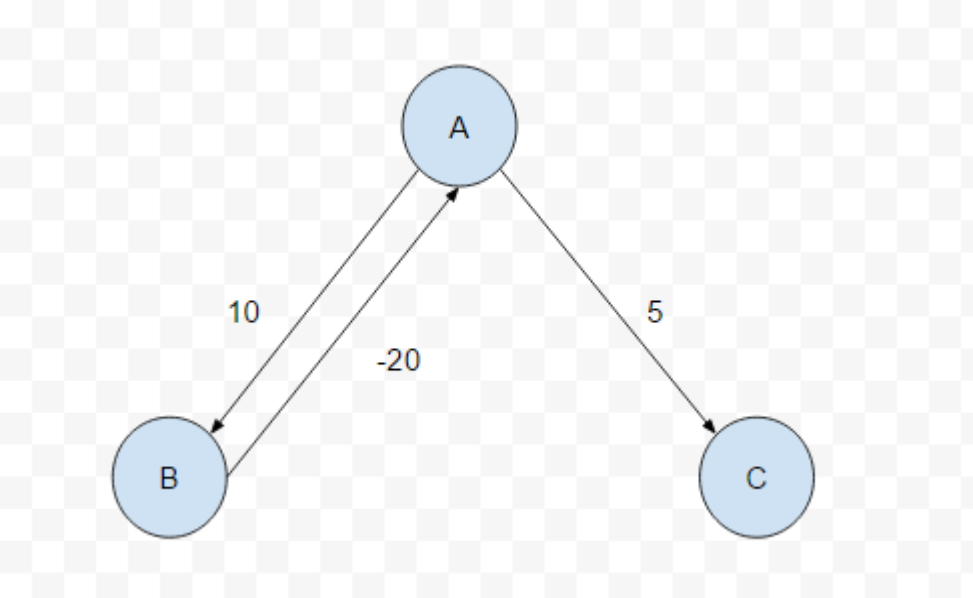
To turn this into a Maximum spanning tree, we simply change the definition of T to be a spanning tree with the maximum possible weight and the ordering of the data structure containing nodes of G.

Applying this to Prims Algorithm, we can initialize all key values to be -1, key of r to be 0 and change Q to be a maximum-priority list, so that the first node in the queue is the one with the maximum weight. We then follow the algorithm by looping until Q is empty by extracting the highest weight from Q. Using this max value node and checking all adjacent nodes, we check if node v of current edge (u,v) is in Q (ie not in tree), and if the edge's weight is greater than v's key.

Correctness: By following same logic of Prim's algorithm, the initial node in A is the root since it has the highest weight in Q. After each iteration, we change the key of each adjacent node currently still in Q to be the weight of the connecting edge, so that on the next pass we can extract the maximum node. Since each node extracted has the highest value that is determined by the highest weighted edge, we build a Maximum spanning tree by continually connecting nodes via the most expensive edge.

Complexity: The algorithm is simply a slightly modified Prim's algorithm, and thus its run time depends on what structure is used to implement Q. Run-time using a Max-heap can yield O((|V| + |E|)log|V|)

Question 8



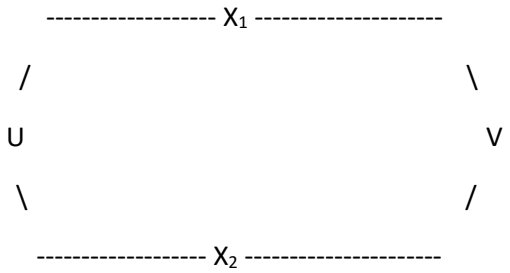
With the above directed graph with a negative cycle, Dijkstra's algorithm will not work. Starting at A, the algorithm will relax (A,C) and (A,B), updating d(C) to 5 and d(B) to 10. Upon extracting the minimum node from Q, C will be returned, from which there are no paths and no updates to distances. Next, B is extracted, but A is in the solution as the root. If we still follow the algorithm defined in the textbook and lecture, we relax A and update its parent to B. The algorithm then terminates as there are no more nodes in Q. This leaves us with pi(B) as A, and pi(C) as A, thus the returned solution for shortest path is (A,C), even though path (A,B,A,C) has a lower weight. This can also cause issues as there is an infinite cycle between A and B, where pi(A) is B and pi(B) is A. In either scenario, the algorithm does not work.

Note: I am unsure if a negative cycle is the answer that is being looked for here, but as far as I can tell from experimenting and reading online discussions/examples, it is impossible to make a counter example without a negative cycle with only 3 nodes if we use the implicit algorithm defined in the textbook and class. If B connected to C with a negative edge, we would still relax C and update d(C) to -10 and pi(C) to B

Question 9

The Floyd-Warshall algorithm works with negative weights. This stems from how the algorithm computes the shortest distance, selecting the minimum weighted path from some node u to v that pass through some node x in the graph. By constantly comparing the paths that include each node, as long as there is no negative cycle (and thus an infinitely small path), the cumulative weight from the least weighted path will be used.

For example, if we have the following:



Path (u,v) that includes X1 can have a negative weight, which lowers the sum of weights on that path, where-as the path with X2 may not. In either case, since we select the minimum path distance between (u,v), it does not matter if one has a negative edge.

Question 10

To have a cycle, we need at least 2 points in a graph, u and v, where there exists paths from (u,v) and (v,u). We know that Floyd-Warshall's algorithm returns the minimum path between all pairs of nodes, so if we first run this on a graph and then run through all node pairs searching for a cycle and recording the lowest weighted cycle, we can find the minimum cycle.

Algorithm: Begin by running Floyd-Warshall's algorithm on the graph. Upon finishing, we have arrays with all the shortest paths between pairs. Next, we would create variables to store the resultant pair and its weight. Next, we would loop through each vertex u and vertex v in the graph and check if there exists paths (u,v) and (v,u). If such a path exists, we have found a cycle, we so we wish to compare its weight to the current minimum weight. If w(u,v) + w(v,u) < current min, replace the current min and update the node pair to u and v.

Correctness: The correctness of Floyd-Warshall is already proven, so provided that it returns the proper shortest paths for pairs and there are no negative cycles, the additional loop checking for cycles and a minimum weight will be correct. By looping through each vertex pair, we check every possible node pair for a cycle. For each cycle found, we compare it to the current known minimum, so even if we find a smaller cycle within a larger cycle (say, path (u,w) contains cycle (u,v), we will still return (u,v)).

Complexity: Runtime of this algorithm is given by runtime Floyd-Warshall + $|V|*|V|*$(2 conditionals + 2 constant operations) = $O(|V|^3) + |V|^2*O(1) = O(|V|^3) + O(|V|^2)$, which gives us $O(|V|^3)$

Question 11

See associated code for Dijkstra's single source SP.