3340 Assignment 3

Matteo Tanzi

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).

1. Describe your algorithm in English (not with pseudo code);

Using the KMP String matching algorithm, we can note that for each matching index of T[i] and P[q + 1] there is in increment of i and q. We can also add to variables prefix_index and prefix_length, and increment each. When a mis match is found, at the index in which the match began, insert the prefix length into some array prefix[], then reset the prefix length to 0. Once the while loop is complete and each prefix length has been inserted into the array, use an efficient approach to find the largest element in the prefix array and print it's index.

2. Show why the algorithm is correct; and

The algorithm is correct because it uses principles from the KMP matching algorithm to ensure that that the characters of a substring match correctly, since the index of the largest prefix is stored and its length is stored also, we can denote that the largest value in the prefix array will store the index of the largest substring prefix match of P.

3. Analyse the complexity of the algorithm

The time complexity of the KMP String Matching Algorithm is O(m + n) and the time complexity to find the largest value in prefix is O(n) meaning the total time complexity is O(m + n + n) = O(n)

2. Give pseudocode to reconstruct and print an LCS from the completed c table and the original sequences X = < x_1,x_2,…x_m > and Y = < y_1, y_2, …, y_n > in O(m+n) time, without using the b table.

```
Reconstruct_LCS(c,X,Y)
    Let n = length of longest common subsequence
    Initialize array print to length n
    Let I = length of X and j = length of Y
    While I and j > 0 do
        If xi = yj then
            S[n] = xi
            n—
            i—
            j—
        else c[i-1,j]>=c[I,j-1] then
            i—
        else
            j—
    print all chars in s
```

3. Describe an efficient algorithm that, given a set $\{x_1, x_2, \ldots, x_n\}$ 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.

1. Describe your algorithm in English (not with pseudo code);

Sort the set of n points such that $X = X_1 <= X_2 <= \ldots X_n$. for each X_n put a closed interval $[x_i, x_i + 1]$ and remove all points, then insert into S. Repeat until X is empty.

2. Show why the algorithm is correct; and

Since the list is in order, the leftmost point will be the smallest point. Each interval from the leftmost point + 1 will fulfill the unit-length closed condition. Since each point is removed and inserted into a list in S. The algorithm is correct.

3. Analyse the complexity of the algorithm

The running time of our algorithm is $O(n \log n + n) = O(n \log n)$, where $O(n \log n)$ is the time for sorting and $O(n)$ is the time for the linear scan.

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.

Choosing two minimal characters C1 and C2 the sum of these frequencies will exceed the frequency of any other character, therefore, via Huffman encoding it will create an internal node with 2 leaves. After this process is continued for all nodes, it is noted that no internal node will have a labal more than 2x any other label, therefore each internal node will create another internal node. The tree produced by Huffman encoding will be a complete binary tree with $2^9 – 1$ internal nodes and a height of 8. Therefore, each codeword will have a length 8, meaning 8 bit fixed length encoding will be the same level of efficiency.

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.

1. Describe your algorithm in English (not with pseudo code);

Set some array knapsack to the size of the capacity of the knapsack + 1. For each value in the array, initialize to 0. Given some value n which is the number of items, some array weight which contains the weight of each item, some array value which contains the value of each item. For each value of I less than capacity, nesting each value of j less than the number of items, where I and j are initialized to 0, set knapsack to the maximum value between its current value and the value of knapsack[i-weight[j] + value[j] if the weight at j is less than i.

2. Show why the algorithm is correct; and

The algorithm is correct because for each capacity and for each item, the value within the item is compared and added to the array, such that the item fulfills the weight constraint

3. Analyse the complexity of the algorithm

The time complexity is O(n*(W + 1)) where w is the capacity and n is the number of items. Meaning it has a worst case time complexity of $O(n^2)$

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

1. Describe your algorithm in English (not with pseudo code);

Modify Prim's minimum spanning tree to find the maximum spanning tree:

Initialiaze some array weights, which will represent the max weight to connect that vertex. Initialize each value with the minimum integer value. Initialize some array parent to track the max span tree, for all unvisited vertices, pick the vertex with the max weight and mark as visted. For each node visited, update the weights of all unvisited adjacent weights by iterating through all the unvisited neighbors of v. For every adjacent vertex x, if the weight of the edge between v and x is greater than the previous value of v, update the value of v with that weight .

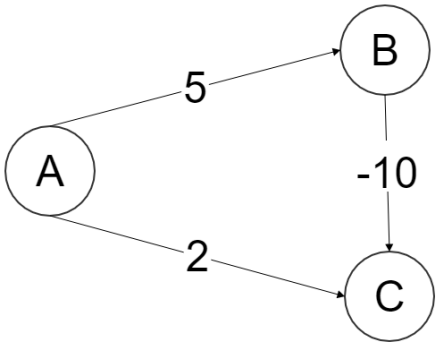
2. Show why the algorithm is correct; and

The algorithm is correct because it uses tools from prims algorithm to ensure that each weight on the graph is a maximum weight, the maximum spanning tree is stored in parent using the maximum weight connecting each vertices to an unvisted vertices.

3. Analyse the complexity of the algorithm

The time complexity of prims algorithm is $O(V^2)$ where V is the number of nodes in the graph

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



When running dijikstra's algorithm:

A = 0; B = ∞; C = ∞;

A = 0; B = min(∞,5) = 5; C = min(∞,2) = 2;

A = 0; B = 5; C = 2;

Since the path from a->c is length 2, we can denote that dijikstra's algorithm is wrong because the true path is -5 from a->b->c

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 $O(|V|^3)$.

1. Describe your algorithm in English (not with pseudo code);

Use floyd-warshall's algorithm to determine the shortest paths between each vertices on the graph. Set some value min to infinity and set some value vertex to none. For each pair of vertices, if the distance of (u,v) plus the distance of (v,u) is less than min, set min to that value, and set the valus in some tuple pair to u and v. Return path(u,v) + path (v+u)

2. Show why the algorithm is correct; and

The algorithm is correct because a path found from u->v and v->u is considered a cycle. Since for each pair of vertices the smallest cycle distance is found and stored, this path can be considered the cycle with the smallest weight.

3. Analyse the complexity of the algorithm

The time complexity of Floyd warshall is $O(|V|^3)$, the time complexity of the for loop that calculates and stores the min cycle is $O(n^2)$ therefore the time complexity is $O(|V|^3)$.