CCCP – TEST 1

Answers By – Tanveer Singh 2020A7PS0084P

Q.1 (a) (c)

Here is a possible dynamic programming solution to this problem, running in O(n^3) time, where n is the length of the string.

Consider an array of booleans dp[n][n][3]. dp[i][j][0] will be True if the substring s[i:j] of the input string (i.e. the substring starting at index i and ending at index j) can return the value "a". Similarly, dp[i][j][1] will be True if s[i:j] can return the value "b", and dp[i][j][2] if it can return the value "c".

Initialise all values in the 3D array to False.

First, we take dp[i][i] = True for its own particular value. For example, in the string "bbbbac", dp[0][0][1] = True, but dp[0][0][0] = dp[0][0][2] = False since s[0] = "b".

So currently, the length of the substrings we were considering was just 1. Before proceeding further, let us make a multiplication function between two entries dp[i][j] and dp[j+1][k], to give the result dp[i][k]. This will depend on the multiplication table that is given to us. For example, since "aa" = "b", dp[i][k][1] will be True if dp[i][j][0] and dp[j+1][k] are both True. (These operations will become easier to handle in code if we convert the symbols into numbers in the beginning, so that a means 0, b means 1, c means 2). So we now have a function multiply(i, j, k) that assigns values for dp[i][k] based on values of dp[i][j] and dp[j+1][k].

Now, let us consider substrings of any length, say, dp[0][k]. We only wish to ascertain whether it is somehow possible to get a/b/c from this substring. So, we test k series of sub-substrings of dp[0][k]. For example, for dp[0][4], we will check it from dp[0][0] + dp[1][4], dp[0][1] + dp[2][4], dp[0][2] + dp[3][4], and dp[0][3] + dp[4][4] using the multiply function multiply(i, j, k).

Finally, we will have dp[0][n-1], the answers for the full substring. For this question, since we want the answer "a", if dp[0][n-1][0] = True, then our answer is Yes, else it is No.

More details on multiply(i, j, k) (see code) – We just want to see if our combined string s[i:k] can give a/b/c or not. For example, lets assume we have dp[0][1][0] = True, dp[0][1][1] = False, dp[0][1][2] = False; dp[2][5][0] = True, dp[2][5][1] = False, dp[2][5][2] = True. We wish to assign dp[0][5]. First, lets check dp[0][1][0] and dp[2][5][0], i.e., for substring values 0 and 0. Since arr[0][0] = 1,

our result will be assigned to dp[0][5][1]. Now, if dp[0][5][1] is already True, then we want to keep it that way, hence the OR operator. If it is False, then dp[0][5][1] will be assigned True if dp[0][2][0] and dp[2][5][0] are both valid, i.e., both True, which is the case in this example. Hence, dp[0][5][1] = True. We do the same for all possible combinations, i.e., 9 combinations.

Q.1 (b) C++ code

```
#include <bits/stdc++.h>
#define MAXN 150
using namespace std;

// Define global variables
bool dp[MAXN][MAXN][3]; //Define MAXN to be a high enough value.
int n;
string s;
// Make global array according to multiplication table. a = 0, b = 1, c = 2.
int arr[3][3] = {{1, 1, 0}, {2, 1, 0}, {0, 2, 2}};

// The multiply function sets dp values.
void multiply(int i, int j, int k) {
for(int a=0; a<3; a++) {
   for(int b=0; b<3; b++) {
      dp[i][k][arr[a][b]] = dp[i][k][arr[a][b]]||(dp[i][j][a]&&dp[j+1][k][b]);
        }
    }
}

int main() {
    // Input string, find its length and initialise dp.
    cin >> s;
    n = s.length();
    for(int i=0; i<n; i++) {
        for(int j=0; j<n; j++) {
            for(int k=0; k<3; k++) {
                dp[i][j][k] = false;
            }
        }
    }
    // First set values for dp[i][i], i.e., individual elements of string.
    for(int i=0; i<n; i++) {
        if(s[i]=='a') dp[i][i][0] = true;
        if(s[i]=='b') dp[i][i][1] = true;
        if(s[i]=='c') dp[i][i][2] = true;
    }
    // Keep increasing the length of substrings and
    // calling multiply for all possible combinations
    // that produce the substrings.
```

```
    for(int l=2; l<=n; l++) {
        // l is the length of the string.
        for(int i=0; i<=(n-l); i++) {
            // i is the starting index.
            // This will give all subtstrings of length l in dp[i][i+l-1].
            for(int k=i; k<(l+i-1); k++) {
                // k is the middle index of strings for multiply function.
                multiply(i, k, i+l-1);
                // This will assign dp[i][i+l1] based
                // on dp[i][k] and dp[k+1][i+l-1].
            }
        }
    }
    if(dp[0][n-1][0]) cout << "YES";
    else cout << "NO";
}
```

Q.1 (c)

Length L varies from 1 to n. For length L, there are N-L+1 substrings to check. Each substring of length L requires L-1 calls to the multiply function (which runs in O(1) time) to be checked. Hence, the running time is –

$$\sum_{l=1}^{n} (n-l+1)(l-1) \approx O(n^3)$$

(Bonus marks for LaTeX??)

Q.2 (a)

```
int occurence(TreeNode* root, int num) {
    if(root!=NULL) {
        if(root->elem==num)
return 1 + occurence(root->left, num) + occurence(root->right, num);
        else
return occurence(root->left, num) + occurence(root->right, num);
    }
    return 0;
}
```

The function proceeds as follows – Firstly, if the Node is NULL, then there is no need to do anything. If the TreeNode exists, however, first check whether its element is equal to num. If it is, add +1 to the answer. After that, check the amount

of times num occurs in the subtrees by calling the function for the left and right subtrees of the node.

Q.2 (b)

```
vector<TreeNode*> rightMostPath(TreeNode* root) {
    vector <TreeNode*> v;
    while(root!=NULL) {
        v.push_back(root);
        root = root->right;
    }
    return v;
}
```

The function proceeds as follows – First we initialise an empty vector v. We will accumulate values in v. Now we enter a loop, which will only terminate when we have reached a null value. At every step, we push_back the pointer to root in the vector, and then change the root pointer to the right subtree of the root. In the end, this will give us a vector of pointers starting from Root of main tree, traversing down right sub-trees and ending at NULL node.

Q.2 (c)

```
bool maxHeap(TreeNode* root) {
    if(root==NULL||root->right==NULL) return true;
    return (root->elem>=root->right->elem&&root->elem>=root->left-
>elem&&maxHeap(root->left)&&maxHeap(root->right));
}
```

The function proceeds as follows – First, we check if the given node is NULL, or whether it is a leaf node. (It is a leaf node if it only points to NULL in right and left). In that case, the subtree with the given node at root will obviously be a max heap. Now, we put 4 conditions –

The node value must be greater than or equal to its right sub-node value.

The node value must be greater than or equal to its left sub-node value.

Its right sub-tree must satisfy this condition.

Its left sub-tree must satisfy this condition.

If all of these conditions are satisfied for the root node, the node is the root of a max heap tree.

Q.3 - Explanation

I extend a formal apology in advance for using one-indexing in this question.

We have a string of n characters, s. Let us first make an array of vectors S of length (n+1) [n+1 as I am using one-indexing], which stores, for each index I, all the words that can begin with the Ith letter of the string s, where I ranges from 1 to n. We don't need the words themselves, we just need to know the index at which the words end.

As an example, take the string "tentative" of length 9. The words starting from the first letter are "ten", "tent", "tentative". We found these words by iterating through the string with a simple for loop and matching each word with the dictionary. These words end and indexes 3, 4, 9. Hence, S[1] = {3, 4, 9}. We can do this for all S in 1 to n, in O(n^2) time.

Now, we make an array of Booleans D of length (n+2). [+1 for one-indexing, and another +1 will be explained later]. Initially, all values will be False, except D[n+1] = True. [Again, this will be explained later]. This array will store, for every index I, *whether it is possible to construct a valid string* (i.e., a string that is a combination of words in the dictionary) *starting from index I or not.* We will iterate through this string in reverse order, i.e., from n to 1. To get any value D[I], we will iterate through S[I]. In S[I], let one of the elements be {J}. Since this means that there is a word starting from I and ending at J, now we just need to check if D[J+1] is a valid string or not, i.e., whether D[J+1] == True. If it is a valid string, we can make D[I] = True. We can check this for all J. Our answer will be the value of D[1] – Yes if True, No if False.

The value D[n+1] has been set to True because if we ever reach "n+1", it means the previous word ended at index n, hence, the whole string starting from index I has been completely filled with valid words.

So, when we are evaluating D[I], all D having index greater than I will already have been evaluated. Let us try to evaluate D[1] for "tentative". S[1] = {3, 4, 9}. So, we check D[3+1], which is False; D[4+1], which is also False; D[9+1], which is True, hence D[1] = True.

D will work in O(n^2) as it will run from n to 1, and for each index I, there may be a maximum of n words starting from it in S[I].

Q.3 – Pseudo Code

```
#include <bits/stdc++.h>
using namespace std;
#define MAXN 1500

// Global Variables
map <string, bool> dict; // Already provided to us.
int n;
```

```
string s;
vector <int> S[MAXN]; //Initialise MAXN to a large enough value.
bool D[MAXN];

int main() {
    // Take length of string and string as input.
    cin >> n >> s;
    s = "0" + s; // For one-indexing :)
    // Populate S with values.
    for(int i=1; i<=n; i++) {
        string temp;
        for(int j=i; j<=n; j++) {
            temp = temp + s[j]; // temp stores the word we need to check.
            if(dict[temp]) S[i].push_back(j);
        }
    }
    // Initialise D.
    for(int i=1; i<=n; i++) D[i] = false;
    D[n+1] = true;
    // Populate D with values, in reverse order.
    for(int i=n; i>=1; i--) {
        for(auto j: S[i]) {
            if(D[j+1]==true) {
                D[i] = true;
                break;
            }
        }
    }
    if(D[1]) cout << "This is a valid string.";
    else cout << "This is not a valid string.";
}
```

Q.4 (a)

Consider the set S of all positive linear combinations of a and b –

S = {au+bv | au+bv > 0; u, v are integers}.

I will attempt to prove that gcd(a, b) is the smallest element of this subset. This will be helpful in proving **Q.4(b)** and will also show that the statement in Q.4(a) is valid.

This set will definitely be non-empty. The proof of that is as follows –

(1) If both a and b are zero, then gcd(a, b) is not even defined so we won't consider this case.
(2) WLOG assume a is non-zero. If a>0, then a.(1) + b.(0) > 0, and if a<0, then a.(-1) + b.(0) > 0.

Hence, this set is always non-empty.

Now, we will try to show that the least element of this set is, in fact, gcd(a, b). [Each non-empty set of real numbers must have a smallest element according to the Well Ordering Principle].

Let us assume that the smallest element in this set is d. Then, d = ax+by for some x and y.

Using the Division Algorithm, we can say that a = qd+r, where q and r are integers and 0 <= r < d. Thus, r = a – qd, that is, r = a – q(ax+by), that is, r = a(1-qx) + b(-qy).

The only possible value of r is 0. Proof by contradiction – If r was positive, it would be an element of the set S, as we are expressing it in the form ax+by. But according to the Division Algorithm, r must be less than d, and d is the least element of S. Hence, r must be 0. Thus, d divides a. Similarly, we can show that d divides b.

So, d is a common divisor of a and b. If we can show that every other common divisor of a and b divides d, we will be able to conclude that d is the greatest common divisor of a and b, as any other common divisor will be less than d. The proof of this is –

Let c be an arbitrary common divisor of a and b. Then, a = cu and b = cv for some integers u,v. Hence, d = ax+by = c(ux + vy), i.e., d is a multiple of c, or, c divides d.

This completes our proof that d = gcd(a, b) = ax+by for some integers x,y.

Q.4 (b)

The Euler Phi function Phi(n) returns the number of positive integers from 1 to n that are relatively prime to n. We wish to prove that the sum of values of Phi(d), where d are the divisors of n, is equal to n itself.

We can split the numbers from 1 to n into sets S(d), where S(d) contains all the numbers m from 1 to n that have gcd(m, n) = d. Obviously, this will cover all divisors of n, as any divisor d of n will have gcd(d, n) = d. Also, each number from 1 to n will go in only one possible set S(d), as there can only be one gcd of a pair of numbers.

We claim that if gcd(m, n) = d, gcd(m/d, n/d) = 1. The proof of this will directly follow from everything we did in **Q.4(a).** If gcd(m, n) = d, mx+ny = d, hence, (m/d)x+(n/d)y = 1. 1 is the smallest possible positive element, so it must be the gcd of m/d and n/d. (This was explained in Q.4(a)).

Now let us go further into any particular set S(d). S(d) contains all values m such that gcd(m/d, n/d) = 1. So, S(d) has all the numbers not exceeding n/d that are relatively prime to n/d. This is the definition of Euler Phi! Hence, the number of elements in S(d) = Phi(n/d).

Since each number from 1 to n comes in any ONE of the sets S(d), n is the sum of Phi(n/d) over all divisors of d.

n = sum(Phi(n/d)) over all divisors d of n.

Now consider, if d is any divisor of n, then its counterpart n/d is also a divisor. So,

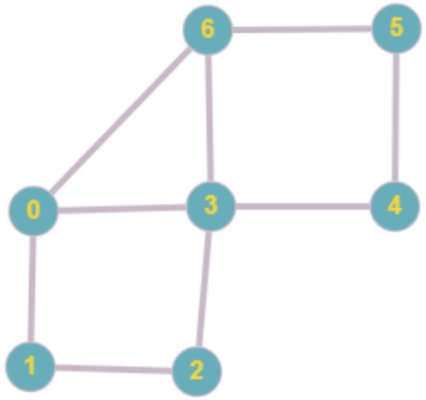
n = sum(Phi(d)) over all divisors d of n.

This completes the proof.

Q.5

(a) DFS will not always work. If we were using DFS, to detect a triangle, we would pass parent and grandparent as argument into the DFS function. DFS will detect a triangle if any node has an edge connecting to its grandparent node. However, in certain cases of graphs with multiple cycles, this algorithm will fail.

(b) Consider a graph –



Here, let us start DFS from 0. We will never be able to find a traversal where a node leads back to its own grandparent. Since, in DFS, we set any visited node to true, and we are going to choose least integer nodes first, here is how the DFS will go – 0→1→2→3→4→5→6. At 3, we will find an edge leading back to 0, but the grandparent of 3 is 1 and not 0. At 6, we find another edge leading back to 0, but the grandparent of 6 is 4 and not 0. Hence, a triangle is never found.

Q.6

Let us create a new graph, with V*(C+1) nodes, where V is the number of cities and C is the max fuel tank capacity of the car. Here, every node will be in the form of a pair (v, c) and will represent the fuel capacity c [Range 0 to C] in a given city v. A node (v, c) will have a directed edge to node (v, c+1), with a cost of p[v] DogeCoins. This edge implies that we are at the city v, our tank currently has c litres of fuel in it and we want to buy an additional litre of fuel using p[v] DogeCoins to give our fuel tank c+1 litres of fuel. Thus, there will be a directed edge from (v,c) to (v,c+1) for every v in V, and c in [0, C-1]. Now consider how cities are connected, i.e., the edges. If there is an edge e from u to v with a distance d in the previous graph, i.e., e = (u, v, d), in our new graph, there will be (c - d) edges linking the different nodes (u, c) to (v, c-d) of the graph. An explanation for this is as follows – Consider a node (u, c). This means that at city u, our car has c litres of fuel. Now, since we know that there is an edge in our original graph from u to v with a distance d, we can use up d litres of fuel to reach the city v, with our tank at c-d litres. So, we go from (u, c) to (v, c-d). We need to do this for all possible values of c/d, such that c and c-d belong to range [0, C]. Also, in our new graph, the edge weights of these cross-linking edges will be 0. This is because we are making our new graph to calculate the DogeCoins spent, not the amount of distance travelled. Now, our graph has V*(C+1) nodes, interconnected. We can now run Djikstra from our starting node (S, 0) to find the smallest cost of reaching any node from it. Our target is city E. To find the minimum cost of reaching city E, we can just take min(E, c) for all c in range 0 to C. This will give us the required answer.

It will take O(V*C + E*C) worst case time to construct the new graph. Running Dijkstra on the new graph will take O(V*C log (V*C)) time.