Part A – Objective (Short Answer)

Question No. 1 – String Matching (4+2+4= 10)

1. What will be the Pi function(described in Knuth Moriss Partt) for given string “ababcababc”. **(4 Marks)**

a	**b**	**a**	**b**	**c**	**a**	**b**	**a**	**b**	**c**
0	**0**	**1**	**2**	**0**	**1**	**2**	**3**	**4**	**5**

2. You are given with pattren p and string s with length of m and n. Let suppose pattren length much smaller than the string. Which algorithm will choose to search pattren in given string. Provide a justification. **(2 Marks)**

   Knuth Moriss Parrt

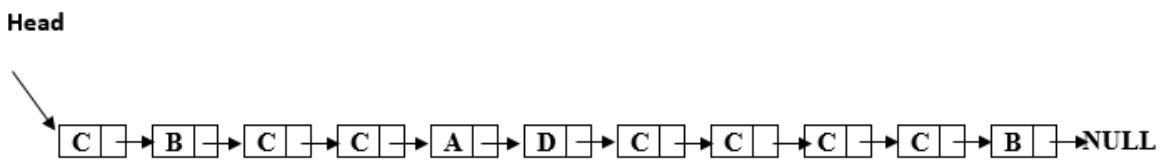
3. Given a pattern "VWX" and using a simple hash function where the hash value of a string is calculated as the sum of ASCII values of its characters modulo a prime number 7, calculate the hash value of the pattern. For the same pattern " VWX " and text "XZTVWXPPP", using the hash function described above, write down the points where we get spurious hit. **(4 Marks**

   Answer: Only Spurious hit will be at ‘PPP’.

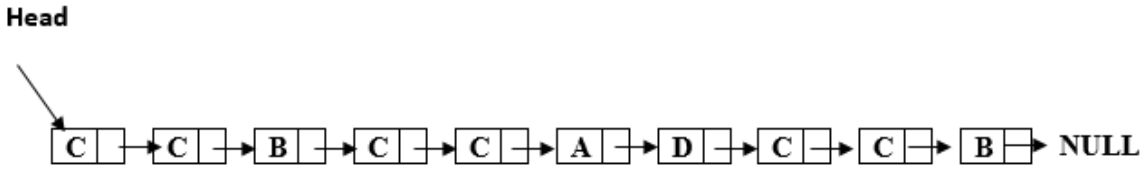
Question No. 2 – Time Complexity Analysis and Amortized Analysis (3+3+4+4+3+3=20 Marks)

```
bool List::Equalize_Occurrences(char d, int maxcount)
{
   Node* ptr = first;
   bool chk=false;
   Node* temp;
   while (ptr != NULL)
   {
      if (ptr->data == d)
      {
         chk = true;
         int count = 0;
         temp = ptr;
         while (ptr != NULL)
         {
            if (ptr->data == d)
            {
               count++;
               ptr = ptr->next;
            }
            else
               break;
         }
         if (count > maxcount)
         {
            while (count > maxcount)
            {
               del_after(temp);
               count--;
            }
         }
         else if (maxcount > count)
         {
            while (count < maxcount)
            {
               ins_after(temp,d);
               count++;
            }
         }  // else
      } // outer if
      ptr = ptr->next;
   } // while
   return chk;
}
```

Consider that a **singly linked list** class, with a head pointer, is already implemented for character datatype. We have to add functionality in the class to balance out the number of consecutive occurrences of a particular character in the list. For that, we have implemented a function **bool Equalize_Occurrences (char key, int maxcount)** of the class list, that takes a character key and maximum count for the consecutive occurrences of the key in parameters. It will then traverse the list, verify, and update the consecutive occurrences of the key according to maximum count and returns true. It returns false if no occurrence of the key is found. For Example, if the singly linked list **L1** contains data as follows:



then after function call **L1.Equalize_Occurrences ('c', 2);** list will be updated as follows.



Let's suppose that there are two built-in functions with constant time complexities;

1. **ins_after**, this function inserts a key value after the node to which ptr is pointing.
2. **del_after,** this function deletes the node after the node to which ptr is pointing.

You are required to answer the following questions:

1. Explain the best, Avg and worst cases for function **Equalize_Occurrences** using Big-O. (**10 Marks**)

   Best Case: O(N) **(3 Marks)**
   Explanation: _____

   Worst Case: O(n x d) **(3 Marks)**

   Explanation: _____

   Average Case: O(n x d) **(4 Marks)**

   Explanation: _____

Amortized Analysis

We have discussed the amortized analysis of dynamic tables in class. Considering the example of a dynamic table, please answer the following questions:

4. In the insertion operation, suppose that instead of doubling the size of the array when it gets full, we increase the size by *4. What will be the amortized cost for a single operation? Prove it using accounting method. **(4 Marks)**

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Size	1	4	4	4	16	16	16	16	16	16	16	16	16	16	16	16	64
Cost	1	2	1	1	5	1	1	1	1	1	1	1	1	1	1	1	17
Amortize	2.5	2.5	2.5	2.5	2.5	2.5	2.5	2.5	2.5	2.5	2.5	2.5	2.5	2.5	2.5	2.5	2.5
Bank	1.5	2	3.5	5	2.5	4	5.5	7	8.5	10	11.5	13	14.5	16	17.5	19	4.5

5. Now, let's analyze the deletion process. When performing deletions, if the table size drops to n/4, we create a new table of half the size and shift the elements into the new array, deleting the previous array. Suppose deletions in a dynamic table can only be performed at the start or the end. Consider the following:

   a. If we always perform deletions at the start, what will be the cost of a single operation? Prove it using the aggregation method. **(3 Marks)**

      Although deletion of one value cost 1 but we have to perform the shift after each deletion which cost will be :
      $1+2+3+4+5+\ldots+n = n(n+1)/2=n^2$ and for single operation n.

   b. If we always perform deletions at the end, what will be the cost of a single operation? Prove it using the aggregation method. **(3 Marks)**

Answer: By using aggregation method, the single operation cost will be c*n which is O(n)

Question No. 3 – Recurrence and Master Theorem Analysis (3+4= 7 Marks)

```
Algorithm ALGO(A, n) // Input: An array A of n real numbers
    if n=1
        return A[0]
    else
        temp ← ALGO(A, n-1)
        if temp ≤ A[ n-1 ]
            return temp
        else
            return A[ n-1 ]
```

a) What does this algorithm compute?

Answer: ALGO computes the smallest element in n-sized array.

b) Write the recurrence for the algorithm and solve it using iteration method.

Answer:

$T(n) = \{ c \text{ if } n=1, T(n-1)+c \text{ otherwise}$

Iterative solution:

$T(n) = T(n-1) + c$

$T(n) = T(n-2) + 2c$

$T(n) = T(n-3) + 3c$

.

.

.

$T(k)= T(n-k) + kc$

Let $n-k = 1$, $k= n-1$

$T(n) = T(1) + (n-1) c$

$T(n) = c+ nc -c$

$T(n) = nc$

Time complexity □ O(n)

Question No. 4 – Dynamic Programming and Divide & Conquer (3+4+5= 12 Marks)

a) Explain under what circumstances dynamic programming technique is better than divide and conquer technique?

Divide-and-conquer algorithms partition the problem into disjoint subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming applies when the subproblems overlap—that is, when subproblems share subsubproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems. A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem.

b) Show why divide and conquer technique is a suitable approach instead of a dynamic programming technique in order to sort numbers using merge sort? Provide your reasoning with the help of a complete example.

Merge Sort cannot use Dynamic Programming, because the subproblems are not overlapping in any way. Following merge sort tree shows each sub problem is distinct.

c) True or False

Sr. No.	Statement	TRUE / FALSE
1.	With all equal-sized intervals, a greedy algorithm based on earliest start time will always select the maximum number of compatible intervals	T
2.	The quicksort algorithm that uses linear-time median finding to run in worst-case O(n log n) time requires Θ(n) auxiliary space.	F
3.	In a connected, weighted graph, every lowest weight edge is always in some minimum spanning tree.	T
4.	For a connected, weighted graph with n vertices and exactly n edges, it is possible to find a minimum spanning tree in O(n) time.	T
5.	In a divide-and-conquer algorithm, the conquer step always involves solving the subproblems directly.	F

Question No. 5 – Greedy (6 Marks)
Consider the pseudo code of Dijkstra algorithm below. Suppose we are using binary heap in to store the nodes. Calculate the worst case time complexity of each fragment considering that given graph is a dense graph.

```
Dijkstra(G)
   for each v ∈ V
      d[v] = ∞;
   d[s] = 0; S = ∅; Q = V;
   while (Q ≠ ∅)
      u = ExtractMin(Q);
      S = S ∪ {u};
      for each v ∈ u->Adj[]
         if (d[v] > d[u]+w(u,v))
            d[v] = d[u]+w(u,v);
```

Relaxation Step

Your Solution here:

EXTRACT_MIN operations takes O(log V) time and there are |V| such operations.
The binary heap can be build in O(V) time.
Operation DECREASE (in the RELAX) takes O(log V) time and there are at most |E| such operations.
Hence, the running time of the algorithm with binary heap provided given graph is sparse is O((V + E) log V).
Note that this time becomes O(E logV) if all vertices in the graph is reachable from the source vertices. For dense graph, it becomes O(V 2 log V).

Question No. 6 – Sorting (4+4+4+4 = 16 Marks)

a) Give at least three reasons as to why someone would use merge sort over quicksort ?

a. Mergesort has Θ(NlogN) worst case runtime versus quicksort's Θ(N2).
b. Mergesort is stable, whereas quicksort typically isn't.
c. Mergesort can be highly parallelized because as we saw in the first problem the left and right sides don't interact until the end. Mergesort is also preferred for sorting a linked list.

b) When will the Quicksort perform poorly (worst case) ?

Quicksort has a worst case runtime of Θ(N2), if the array is partitioned very unevenly at each iteration.

c) We have a system running insertion sort and we find that it's completing faster than expected. What could we conclude about the input to the sorting algorithm?\

Answer: Input already sorted.

d) Explain which sorting algorithm you would use to sort the input array the fastest and why you chose this sorting algorithm. An array of n Comparable objects that is sorted except for k randomly located elements that are out of place (that is, the list without these k elements would be completely sorted)

Sort: Insertion sort

- Runtime: O(nk)
- Explanation: For the n − k sorted elements, insertion sort only needs 1 comparison to check that it is in the correct location (larger than the last element in the sorted section). The re-maining k out-of-place elements could be located anywhere in the sorted section. In the worst case, they would be inserted at the beginning of the sorted section, which means there are O(n) comparisons in the worst-case for these k elements. This leads to an overall runtime of O(nk + n), which simplifies to O(nk).

Question No. 7 – Heaps and Hashing (2+2+4+2+4+2=14 Marks)

a) Does this array represent a Max-Heap with the root at index 1? (Ignore the data at index 0.)

index	0	1	2	3	4	5	6	7	8	9	10	11	12
Data	12	78	50	30	8	65	22	28	4	6	12	2	13

Answer: Not a heap.

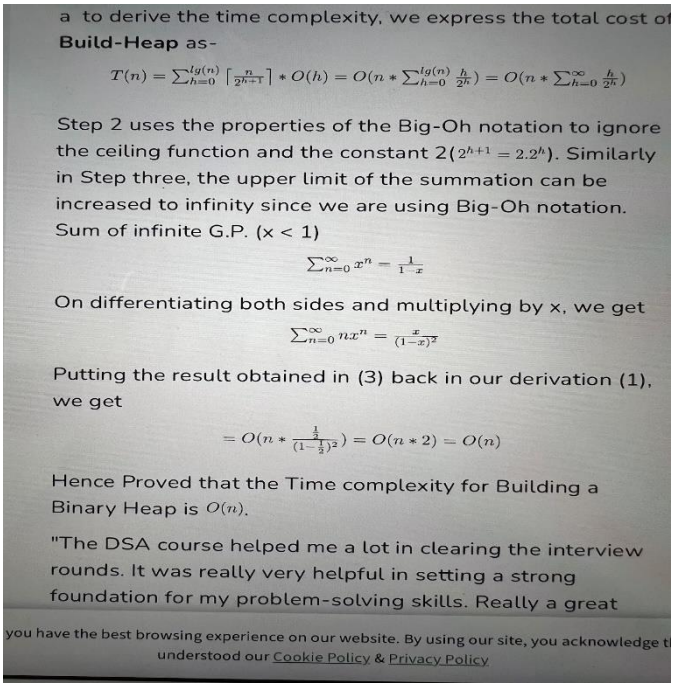
b) Consider 21 20_a 20_b 12 11 8 7 elements , show that whether the Heapsort is stable or not ?

Note: A sorting algorithm is stable if identical elements remain in their original order after sorting.

Answer: Stability for sorting algorithms mean that if two elements in the list are defined to be equal, then they will retain their relative ordering after the sort is complete. Heap operations may mess up the relative ordering of equal items and thus is not stable.

c) Consider the following algorithm for building a Heap of an input array A. Solve in step by step the time complexity of building a heap function below:

```
BUILD-HEAP(A)

  heapsize := size(A);

  for i := floor(heapsize/2) downto 1

    do HEAPIFY(A, i);

  end for

END
```



a to derive the time complexity, we express the total cost of **Build-Heap** as-

$$T(n) = \sum_{h=0}^{lg(n)} \lceil \frac{n}{2^{h+1}} \rceil * O(h) = O(n * \sum_{h=0}^{lg(n)} \frac{h}{2^h}) = O(n * \sum_{h=0}^{\infty} \frac{h}{2^h})$$

Step 2 uses the properties of the Big-Oh notation to ignore the ceiling function and the constant 2($2^{h+1} = 2.2^h$). Similarly in Step three, the upper limit of the summation can be increased to infinity since we are using Big-Oh notation.

Sum of infinite G.P. (x < 1)

$$\sum_{n=0}^{\infty} x^n = \frac{1}{1-x}$$

On differentiating both sides and multiplying by x, we get

$$\sum_{n=0}^{\infty} nx^n = \frac{x}{(1-x)^2}$$

Putting the result obtained in (3) back in our derivation (1), we get

$$= O(n * \frac{\frac{1}{2}}{(1-\frac{1}{2})^2}) = O(n * 2) = O(n)$$

Hence Proved that the Time complexity for Building a Binary Heap is $O(n)$.

"The DSA course helped me a lot in clearing the interview rounds. It was really very helpful in setting a strong foundation for my problem-solving skills. Really a great

Hashing

d) We have seen that under the assumption of uniform hashing, collision resolution by chaining, and constant time computable hash function, deletion and search are both O(1 + n /m) in the average case. We have also seen that this can in fact be reduced to O(1) under one additional assumption. Describe this additional assumption in one sentence.

n<= m

e) Which one of the following hash functions on integers will distribute keys most uniformly over 10 buckets numbered 0 to 9 for i ranging from 0 to 2020?

i. (A) h(i) =i^2 mod 10
ii. (B) h(i) =i^3 mod 10

iii. (C) h(i) = (11 ∗ i^2) mod 10
iv. (D) h(i) = (12 ∗ i) mod 10

Part B

f) Explain the concept of double hashing and why do need it? One side effect?

Double Hashing is a collision resolution technique in open addressing that uses two hash functions to determine the probe sequence. The first hash function determines the initial position, and the second hash function determines the step size for probing

Part B - Dynamic Programming

Question No. 8 – LCS (12+5+3= 20)

Apply LCS bottom-up approach on given strings.

String A: "abracadabra",

String B: "cadabracar"

Marks distribution:

- Correct Answer: 12 Marks
- Solution(Track): 5 Marks
- Backtrack Algorithms worst case time complexity:O(n+m) 3 Marks.

No cutting and overwriting, otherwise marks will be deducted.

		a	b	r	a	c	a	d	a	b	r	a
	-1	0	1	2	3	4	5	6	7	8	9	10
-1	0	0	0	0	0	0	0	0	0	0	0	0
c	0	0	0	0	0	1	1	1	1	1	1	1
a	1	0	1	1	1	1	2	2	2	2	2	2
d	2	0	1	1	1	1	2	3	3	3	3	3
a	3	0	1	1	1	2	2	3	4	4	4	4
b	4	0	1	2	2	2	2	3	4	5	5	5
r	5	0	1	2	3	3	3	3	4	5	6	6
a	6	0	1	2	3	4	4	4	4	5	6	7
c	7	0	1	2	3	4	5	5	5	5	6	7
a	8	0	1	2	3	4	5	6	6	6	6	7
r	9	0	1	2	3	4	5	6	6	6	7	7

Question No. 9 – Matrix Multiplication(5 Marks)

Show the Brackets for the set of matrixes by looking at table.
Matrix: A, B, C, D, E
D={3, 8, 1, 2, 5, 7}
I prefer that you present the solution in tree form, although you are allowed to represent it in other forms.

Optimal Cost Matrix	Solution Matrix	Your Solution.

	1	2	3	4	5	6
1	0	24	30	49	90	129
2		0	16	50	101	135
3			0	10	45	87
4				0	70	154
5					0	210
6						0

	A	**B**	**C**	**D**	**E**
A	**0**	**1**	**1**	**2**	**2**
B		**0**	**2**	**3**	**2**
C			**0**	**3**	**3**
D				**0**	**4**
E					**0**

The cheapest method to compute ABCDE is ((AB)((CD)E)) with cost 90:

Question No. 10 – OBST(20 Marks)

Determine the cost and structure of an optimal binary search tree for a set of n =5 keys with the following probabilities:

i	0	1	2	3	4	5
p_i		0.05	0.15	0.10	0.20	0.10
q_i	0.10	0.05	0.05	0.05	0.05	0.10

You are required to fill in the following tables of **'w', 'e'** and **'root'** and show the calculations in the given space below.

Table 'e' (8 Marks)

	0	**1**	**2**	**3**	**4**	**5**
1	0.10	0.35	0.80	1.20	1.95	2.6
2		0.05	0.35	0.75	1.40	2.0
3			0.05	0.30	0.80	1.35
4				0.05	0.40	0.95
5					0.05	0.40
6						0.10

Table 'w' (6 Marks)

	0	**1**	**2**	**3**	**4**	**5**
1	0.10	0.20	0.40	0.55	0.80	1.00
2		0.05	0.25	0.40	0.65	0.85
3			0.05	0.20	0.45	0.65
4				0.05	0.30	0.50
5					0.05	0.25
6						0.10

Table 'root' (6 Marks)

	1	**2**	**3**	**4**	**5**
1	1	2	2	2	4
2		2	2	3	4
3			3	4	4
4				4	4
5					5

Question No. 11 – Dynamic Programming (20 Marks)
Given an array of non-negative integers and an integer sum. We have to tell whether there exists any subset in an array whose sum is equal to the given integer sum. (10 marks)

Examples:

Input: arr[] = {3, 34, 4, 12, 3, 2}, sum = 7
Output: True
Explanation: There is a subset (4, 3) with sum 7.

a) Solve the subset sum problem using recursion, Write the recursive function.(10 marks)

b) Solve the subset sum problem using Tabular/Dynamic programing, write the code for dynamic programming solution. (10 marks)

```
#include <bits/stdc++.h>
using namespace std;
// Returns true if there is a subset
// of set[] with sum equal to given sum
bool isSubsetSum(int set[], int n, int sum)
{
        // Base Cases
        if (sum == 0)
    return true;
        if (n == 0)
    return false;



        // If last element is greater than sum,
        // then ignore it
        if (set[n - 1] > sum)
    return isSubsetSum(set, n - 1, sum);
```

```
        // Else, check if sum can be obtained by any
        // of the following:
        // (a) including the last element
        // (b) excluding the last element
        return isSubsetSum(set, n - 1, sum)
            || isSubsetSum(set, n - 1, sum - set[n - 1]);
}


// Driver code
int main()
{
        int set[] = { 3, 34, 4, 12, 5, 2 };
        int sum = 9;
        int n = sizeof(set) / sizeof(set[0]);
        if (isSubsetSum(set, n, sum) == true)
    cout << "Found a subset with given sum";
        else
    cout << "No subset with given sum";
        return 0;
}
```

b)
```
// CPP program for the above approach
#include <bits/stdc++.h>
using namespace std;


// Taking the matrix as globally
int tab[2000][2000];


// Check if possible subset with
// given sum is possible or not
int subsetSum(int a[], int n, int sum)
{
        // If the sum is zero it means
        // we got our expected sum
        if (sum == 0)
    return 1;

        if (n <= 0)
    return 0;

        // If the value is not -1 it means it
        // already call the function
        // with the same value.
        // it will save our from the repetition.
        if (tab[n - 1][sum] != -1)
    return tab[n - 1][sum];
```

```
        // If the value of a[n-1] is
        // greater than the sum.
        // we call for the next value
        if (a[n - 1] > sum)
    return tab[n - 1][sum] = subsetSum(a, n - 1, sum);
        else
        {
    // Here we do two calls because we
    // don't know which value is
    // full-fill our criteria
    // that's why we doing two calls
    return tab[n - 1][sum] = subsetSum(a, n - 1, sum) ||
                    subsetSum(a, n - 1, sum - a[n - 1]);
        }
}



// Driver Code
int main()
{
        // Storing the value -1 to the matrix
        memset(tab, -1, sizeof(tab));
        int n = 5;
        int a[] = {1, 5, 3, 7, 4};
        int sum = 12;



        if (subsetSum(a, n, sum))
        {
    cout << "YES" << endl;
        }
        else
    cout << "NO" << endl;
```

Question No. 12 – Dynamic Programming(10 Marks)

Given a string **S**, find the common palindromic sequence (A sequence that does not need to be contiguous and is a palindrome), which is common in itself. You need to return the length of the longest palindromic subsequence in A.

A string is said to be palindrome if the reverse of the string is the same as the string. For example, “abba” is a palindrome, but “abbc” is not a palindrome.

Examples:

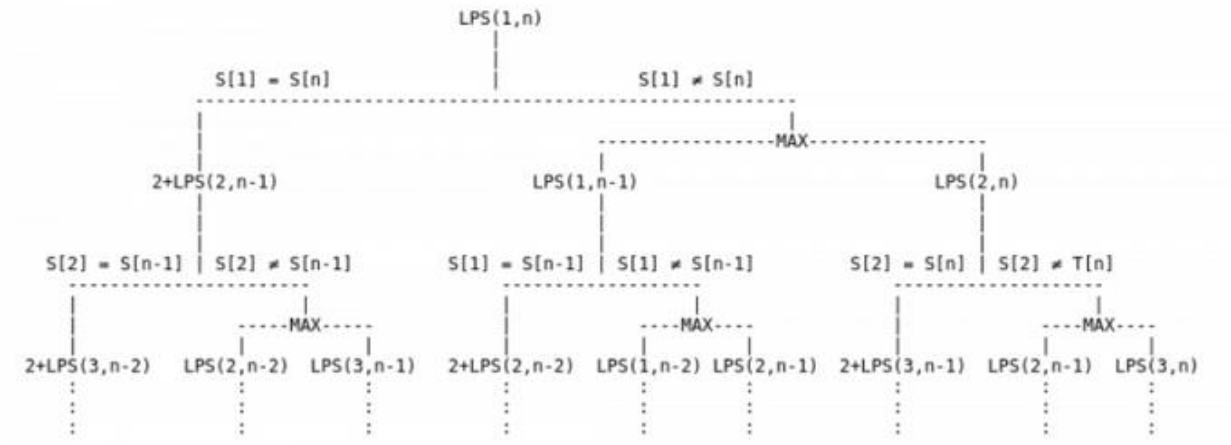
Input: S = “BEBEEED”
Output: 4
Explanation: The longest common palindromic subsequence is “EEEE”, with a length of 4 .

Brute force: Recursion Approach : A simple approach to solve this problem is to generate all the subsequences of the given string and find the longest palindromic string among all the generated strings. There are a total of 2^N strings possible, where N denotes the length of the given string.

The idea is to check and compare the first and last characters of the string. There are only **two** possibilities for the same:

- If the first and the last characters are the same, it can be ensured that both the characters can be considered in the final palindrome and hence, add **2** to the result, since we have found a sequence of length **2** and recurse the remaining substring **S[i + 1, j – 1]**.
- In case, the first and the last characters aren’t the same, the following operation must be performed:
    - Recurse **S[i + 1, j]**
    - Recurse **S[i, j – 1]**
- Find the maximum length obtained amongst them.

The recursion tree can be shown as follows: **LPS(1, N)** denotes the length of the longest palindromic subsequence of the given string **S[1..N].**



Now that you know the recursive solution , after identifying the overlapping subproblem, construct the dynamic programming solution of the above problem, clearly show how the subproblems are stored .

Your Answer:

```
public int LPS(string S) {
```

```
    int dp[S.length()][S.length()];

    for (int i = S.length() - 1; i >= 0; i--) {
        dp[i][i] = 1;
        for (int j = i+1; j < S.length(); j++) {
            if (S[i] == S[j]) {
                dp[i][j] = dp[i+1][j-1] + 2;
            } else {
                dp[i][j] = max(dp[i+1][j], dp[i][j-1]);
            }
        }
    }
    return dp[0][S.length()-1];
}
```