DSA
Assignment 1

MM: 40

Instructions:-

Section 1

Q1.

MM : 3+2+2=7

1. Write a code for the Tower of Hanoi Algorithm using two temporary pegs I.e. There is a source pole/peg named T1, two temporary pegs named T2 and T3 ; a destination pole/peg named T4. Take the number of disks as input from the user.

Solution:-

```
#include <iostream>
//give 0 marks if the code dosent compile.
void move(int n, char *src, char *dst, char *tmp1, char *tmp2) {

    if (n == 0) {
        return;
    }
    if (n == 1) {
        std::cout <<src << "->" << dst << std::endl;
        return;
    }
    if( n ==2 ){
        std::cout  << src << "-> " << tmp1 << std::endl;
        std::cout <<src << "->" << dst << std::endl;
        std::cout <<tmp1 << "->" << dst << std::endl;
        return ;
    }
```

```
// give total 1 mark if the base cases are mentioned.
    move(n-2, src, tmp1, dst, tmp2);
//.5 marks for the above recursive move statement.
    std::cout <<src << "->" << tmp2 << std::endl;
    std::cout <<src << "->" << dst << std::endl;
    std::cout <<tmp2 << "->" << dst << std::endl;
//.5 marks for the above statements
    move(n-2, tmp1, dst, src, tmp2);
//.5 marks for the above recursive move statement.
}

int main() {
    // Example usage
    char src[] = "T1";
    char tmp1[] = "T2";
    char tmp2[] = "T3";
    char dst[] = "T4";

    int num_disks ;
    std :: cin >> num_disks;
    move(num_disks, src, dst, tmp1, tmp2);

    return 0;
}
//.5 for the correct main class
// code for traditional hanoi for 1.3
#include <iostream>

void move(int n, char *src, char *dst, char *tmp1 ) {
    if (n == 1) {
        std::cout << src << " -> " << dst << std::endl;
        return;
    }

    move(n - 1, src, tmp1, dst);
    std::cout  << src << " -> " << dst << std::endl;
    move(n - 1, tmp1 , dst , src);
}

int main() {
    // Example usage
    char src[] = "T1";
    char tmp1[] = "T2";
```

```
//    char tmp2[] = "T3";
    char dst[] = "T3";

    int num_disks ;
    std :: cin >> num_disks;
    move(num_disks, src, dst, tmp1);

    return 0;
}
```

2. Give the sequence of moves required to move 8 disks from source to destination, using the above algorithm. Explain how the code is working with 8 disks.

Output for 5 Modified Hanoi Input 5 Output

```
T1 -> T4, T1 -> T3, T1 -> T2, T3 -> T2, T4 -> T2, T1 -> T3, T1 -> T4,
T3 -> T4, T2 -> T1, T2 -> T3, T2 -> T4, T3 -> T4, T1 -> T4
```

// 0 marks if the steps are same as that for 1 temp peg.
// +1 if upto max 10 steps are missing.

3. Compare the answer you have received in Q1.2 with the answer you will receive with the traditional Tower of Hanoi Setup (1 source pole, 1 temporary pole, 1 destination pole)I.e. compute and compare their time complexities.

Traditional TOH:-

```
T1 -> T3, T1 -> T2, T3 -> T2, T1 -> T3, T2 -> T1, T2 -> T3, T1 -> T3, T1 -
> T2, T3 -> T2, T3 -> T1, T2 -> T1, T3 -> T2, T1 -> T3, T1 -> T2, T3 ->
T2, T1 -> T3, T2 -> T1, T2 -> T3, T1 -> T3, T2 -> T1, T3 -> T2, T3 -> T1,
T2 -> T1, T2 -> T3, T1 -> T3, T1 -> T2, T3 -> T2, T1 -> T3, T2 -> T1, T2 -
> T3, T1 -> T3
```

The major difference is that we are able to move destination in each call by not going against the rule of Tower of Hanoi(of putting a larger disc on a smaller disc) , and then we only have to move n-2 from a temporary tower to destination using source and one support tower for modified algo . Since there is 1 less temporary towers in original, which is why we have more function calls in original tower of hanoi as compared to modified one , since modified algo has one more support tower.
Time complexity of Original tower of Hanoi (Src, dst, tmp1) = O(2^n)
Time complexity of modified hanoi (Src, dst, tmp1, tmp2) = O(2^(n/2))

// 1 mark for comparing the number of moves from the traditional TOH and the modified TOH
// 1 mark for comparing the time complexity.

Output format for Q1.1
>> Enter the number of disks :1
>>The sequence of steps are:-
T1 -> T4

>> Enter the number of disks : 2
>>The sequence of steps are:-
T1 -> T2
T1 -> T4
T2 -> T4

>> Enter the number of disks : 3
>>The sequence of steps are:-
T1 -> T2
T1 -> T3
T1 -> T4
T3 -> T4
T2 -> T4

Q2.

MM: 2+1=3

1. Write a recursive and an iterative code for Merge sort. Take the array to be sorted as input from the user. Assume it to be an integer array only. Print the sorted array from both recursive call and iterative call.
2. Explain the iterative code by an example.

Output format for Q2.1
>> Enter the array to be sorted with elements separated by comma: 32,42,1,4,32,15,6
The sorted array by Recursion is :
1,4,6,15,32,42
The sorted array by Iteration is :
1,4,6,15,32,42

Solution:-
The merge sort solutions have been shared and taught in Tutorials.
A good resource on it:- https://www.baeldung.com/cs/non-recursive-merge-sort
// 1 mark for correct recursive code
// 1 mark for correct iterative code.
// 1 mark if the iterative code is explained using a dry run

Q3 **MM : 2+3=5**

a) **Shinigami took Light Yagami's special notebook and issued a challenge. To get it back, Light needs to solve the "Min Max" problem. Light's task: Given an array of 'N' integers, determine the maximum values obtained by taking the minimum element over all possible subarrays of varying sizes from 1 to 'N'.**

Consider an array, Arr = [3, 1, 2, 4].

For subarrays of size 1: max(min(3), min(1), min(2), min(4)) = 4
For subarrays of size 2: max(min(3, 1), min(1, 2), min(2, 4)) = 2
For subarrays of size 3: max(min(3, 1, 2), min(1, 2, 4)) = 1
For subarrays of size 4: max(min(3, 1, 2, 4)) = 1

The resulting array is [4, 2, 1, 1].

Solution:-
We will use two nested loops for sliding on the window(subarrays) of every possible size and one more inner loop to traverse the window and store the minimum element of the current window in a 'temp' variable.

We will create an array named 'answer'. The 'answer[i]' will store the maximum of all the available minimum of every window size 'i'.

If ‘i’ and ‘j’ are the starting and ending indexes of the window, then its length = j-i+1.

So we update our ‘answer[length]’ with the maximum of all the available minimum of every window size ‘i’ with the help of a ‘temp’ variable

i.e, ‘answer[length]’ = max(answer[length] , temp).

Time Complexity O(N ^ 3),
where ‘N’ is the number of elements present in the array. We ran 3 nested loops in total, 2 for sliding over the window and one for calculating the answer for the current window. Hence, the overall Time Complexity is O(N ^ 3).

```
//give 0 marks if the code dosent compile.
vector<int> maxMinWindow(vector<int> arr, int n)
{
    // Definition: answer[i] will store the maximum of minimum of every
window of size 'i'.
    vector<int> answer(n);
```

```
      for (int i = 0; i < n; ++i)
      {
        answer[i] = INT_MIN;
    }//.25 marks for correct initialization

    // Sliding on all possible windows
    for (int i = 0; i < n; ++i)
      {
        for (int j = i; j < n; ++j)
            {
//.25 for using nested loops

            // Start is the starting index of current window
            int start = i;

            //  End is the ending index of current window
            int end = j;

            // Temp will store minimum element for the current window
            int temp = INT_MAX;
            for (int k = start; k <= end; ++k)
                  {
                temp = min(temp, arr[k]);
            }
//.5 for the correct implementation of the code that find the min element
in each window.

                  int length = end - start;

            // Update the answer of current window length
            answer[length] = max(answer[length], temp);
//.5 for answer vector at index of the current window length with the max
of all the min elements found.
        }
    }

    return answer;
//.5 for finding the max of answer vector to return the final answer.
}
```

b) (3 marks) Help Light Yagami to solve this problem in O(n) time.

Solution:-

- **We will follow some reverse strategy for building our final solution instead of finding minimums for every possible window. We will find how many windows our current element can be the answer.**
- **To find that we will calculate the indexes of the next smaller and previous smaller element for every element given in the array. The next smaller element is some number that is smaller than the current element and lies nearest on the right-hand side of the current element. Similarly, the previous smaller is some number that is smaller than the current element and lies nearest on the left-hand side of the current element.**
- **If there is no next smaller element for the current number then we will assume its index having value N and if there is no previous smaller element for the current number then we will assume its index having value -1.**
- **The above two arrays of the next smaller and previous smaller can be done in linear time with the help of Monotonic Stack.**
- **Suppose 'next[i]' = index of next smaller element, 'prev[i]' = index of previous smaller element. Now, we know the 'ARR[i]' will be the minimum of the window of size -> size = next[i] - prev[i] + 1. So, we will directly use the value of 'ARR[i]' for updating the answer of window having size = next[i] - prev[i] + 1.**
- **After doing this for every element we can update our answer for windows of some different lengths i.e, we are still left with some window sizes for which the answer is not calculated yet. So to fill the remaining entries we will use some observations listed below:-**
    - **The answer for some window having size 'L' would always be greater than or equal to the answer for a window having a length greater than L i.e, L+1, L+2 .... so on.**
    - **Hence, we will update the remaining uncalculated answer for some windows having length 'L' with the maximum suffix starting from length 'L+1'.**

Time Complexity O(N), where 'N' is the number of elements present in the array. The Time Complexity for creating the next array and the prev array from Monotonic Stack takes linear time. For each element, we update our answer using next and prev in constant time. Hence, the overall Time Complexity is O(N).

```
//give 0 marks if the code dosent compile.
#include <stack>

// This function will return an array.
// Each ith position contains the index of previous smaller elements in the
original array.
vector<int> previousSmaller(vector<int> &arr, int n)
{
    vector<int> prev(n);
    stack<int> s;
//+.5 for using stack
    for (int i = 0; i < n; i++)
    {
        while (!s.empty() && arr[s.top()] >= arr[i])
        {
            s.pop();
        }

        if (s.empty())
        {
            prev[i] = -1;
        }
        else
        {
            prev[i] = s.top();
        }

        s.push(i);
    }
//+1 for the correct implementation of the loop above.

    return prev;
}

// This function will return an array.
// Each ith position contains the index of next smaller elements in the
original array.
vector<int> nextSmaller(vector<int> &arr, int n)
{
    stack<int> s;
    vector<int> next(n);

    for (int i = n - 1; i >= 0; i--)
    {
        while (!s.empty() && arr[s.top()] >= arr[i])
```

```
        {
            s.pop();
        }

        if (s.empty())
        {
            next[i] = n;
        }
        else
        {
            next[i] = s.top();
        }

        s.push(i);
    }

    return next;
}
//+.5 for the above function

vector<int> maxMinWindow(vector<int> &arr, int n)
{
    // Definition: answer[i] will store the maximum of minimum of every window
of size 'i'.
    vector<int> answer(n, INT_MIN);

    // Definition: next[i] will store the index of next smaller element which
lie on the right hand side of 'i'.
    vector<int> next = nextSmaller(arr, n);

    // Definition: prev[i] will store the index of previous smaller element
which lie on the left hand side of 'i'.
    vector<int> prev = previousSmaller(arr, n);

    for (int i = 0; i < n; i++)
    {
        // Length of the window in which a[i] is minimum
        int length = next[i] - prev[i] - 1;

        // Update the answer[length-1] ( 0 based indexing )  with a[i]
        answer[length - 1] = max(answer[length - 1], arr[i]);
    }

      // Some entries in answer[] may not be filled yet.
      // We fill them by taking maximum element from suffix starting from 'i'.
```

```
    for (int i = n - 2; i >= 0; i--)
    {
        answer[i] = max(answer[i], answer[i + 1]);
    }

    return answer;
}
//+1 for the function above.
```

Section 2

---

1. (15 points) For the given three sorting algorithms: insertion sort, selection sort and merge sort select the most suitable sorting algorithm for the scenarios provided below and provide a rationale for your choice. Your justification carries more weight than your selection. Each sorting algorithm might be used more than once. If multiple sorting algorithms could fit a scenario, outline their advantages and disadvantages for each of them, and then pick the one that aligns best with the application. Clearly state and justify any assumptions you make and perform asymptotic analysis. "Best" should be evaluated based on **asymptotic running time**.

For each part :
2.5 marks : for correct explanation
2.5 marks : for correct sorting algorithm

(a) (5 marks) You have a custom data structure, *DS*1, that organizes *k* items andsupports two operations:
DS1.get_at_index(j) **takes constant time, and**
DS1.set_at_index(j, x) **takes Θ(*k* log *k*) time.**
Your task is to sort the items in DS1 in-place.

Solution: This part requires an in-place sorting algorithm, so we cannot choose merge sort, as merge sort is not in-place. Insertion sort performs $O(n^2)$ get at and $O(n^2)$ set at operations, so would take $O(n^3 \log n)$ time with this data structure. Alternatively, selection sort performs $O(n^2)$ get at operations but only $O(n)$ set at operations, so would take at most $O(n^2 \log n)$ time with this data structure, so we choose selection sort.

(b) (5 marks) Imagine you possess a fixed array A containing references to n comparable objects, where comparing pairs of objects takes $\Theta(\log n)$ time. Select the most suitable algorithm to sort the references in A in a manner that ensures the referencedobjects appear in non-decreasing order.

Solution: For this problem, reads and writes take constant time, but comparisons are expensive, $O(\log n)$. Selection and insertion sorts both perform $O(n^2)$ compar- isons in the worst case, while merge sort only performs $O(n \log n)$ comparisons, so we choose merge sort.

(c) (5 marks) Suppose you are given a sorted array A containing n integers, eachof which fits into a single machine word. Now, suppose someone performs some $\log \log n$ swaps between pairs of adjacent items in A so that A is no longer sorted. Choose an algorithm to best re-sort the integers in A.

Solution: The performance of selection sort and merge sort do not depend on the input; they will run in $\Theta(n^2)$ and $\Theta(n \log n)$ time, regardless of the input. Insertion sort, on the other hand, can break early on the inner loop, so can run in $O(n)$ time on some inputs. To prove that insertion sort runs in $O(n)$ time for the provided inputs, observe that performing a single swap between adjacent items can change the number of inversions in the array by at most one. Alternatively, every time insertion sort swaps two items in the inner loop, it fixes an inversion. Thus, if an array is k adjacent swaps from sorted, insertion sort will run in $O(n + k)$ time. For this problem, since $k = \log \log n = O(n)$, insertion sort runs in $O(n)$ time, so we choose insertion sort.

2. (10 points) (a) (5 marks) Implement stack and queue from scratch using pointers in C++. They should have all the functions supported by stacks and queues, e.g., insert, pop, top, etc. (Plagiarism will be strictly checked.)

Solution: Check their stack and queue implementation. They shouldn't use an array for storing elements. They must use dynamic memory allocation (using point- ers), node class, or struct. Run the code and perform all standard operations like push(), pop(), and empty(). If all works fine, give full marks. Similarly, check the implementation of the queue using dynamic memory allocation.

(b) (5 marks) Use your stack implementation from above and solve the following problem:- Given a circular integer array nums (i.e., the next element of nums[nums.length - 1] is nums[0]), return the next greater number for every element in nums. The next greater number of a number x is the first greater number to its traversing order next in the array, which means you could search circularly to find its next greater number.

Input: [1, 2, 3, 4, 3]
Output: [2, 3, 4, -1, 4]

Solution: This is a modification of the standard question (next Greater element using stack.) Note:- Students must use their implementation of stack, not inbuilt implemen- tation. Time and Space complexity should be O(n).

```
vector<int> nextGreaterCircularArray(vector<int>& nums) {
    int n = nums.size();
    vector<int> result(n, -1); // Initialize result array with -1
    stack st;  // students implementation of stack

    // Traverse the array twice to handle circularity
    for (int i = 0; i < 2 * n; ++i) {
        int index = i % n;
        while (!st.empty() && nums[st.top()] < nums[index]) {
            result[st.top()] = nums[index];
            st.pop();
        }
        if (i < n)
            st.push(i);
    }

    return result;
}
```