DSA-ASSIGNMENT

SECTION-A

Ques-1 (a). 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.

Ans-1(a). MT24115_MT24110_Sec1_Q1.cpp

Ques-1(b). Give the sequence of moves required to move 5 disks from source to destination, using the above algorithm. Explain how the code is working with 5 disks.

Ans-1(b).

```
O(2^(n/2)): We moves two disk at a time;
towerofhonoi (nDisk, source, tempPeg1, tempPeg2, destnation):
check nDisk is 0:
    return

check nDisk is 1:
    show source --> destination
    return
othherwise
    towerofhonoi (nDisk-2, source, destination,tempPeg2,tempPeg1)
    show source --> tempPeg2
    show source --> destination
    show tempPeg2 --> destination
    towerofhonoi (nDisk-2, tempPeg1, source, tempPeg2,dstnation)
```

Enter the number of disks: 5

- T1 --> T3
- T1 --> T4
- T1 --> T2
- T4 --> T2
- T3 --> T2
- T1 --> T3
- T1 --> T4
- T3 --> T4
- T2 --> T1
- T2 --> T3
- T2 --> T4
- T3 --> T4
- T1 --> T4

Ques-1(c). 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, temporary pole, 1 destination pole) I.e. compute and compare their time complexities.

Ans-1(c). We have computed the complexity ***2^n*** and ***2^(n/2)*** while using 3 pegs and 4 pegs respectively.

- → T(n) = 2T(n-1) + k eq—1
- → T(n-1) = 2T(n-2) + k eq—2
- → By putting eq-2 into eq-1
- → T(n) = 2(2T(n-2) + k) + k
- → T(n) = 4T(n-2) + 3k
- → To solve equation until get T(1)
- → We conclude
- → T(n) = 2^m(T(n-m) + 2^(m-1)k + k eq—3
  - ○ n-m = 0 (only when 2m will be equal to n)
- → That's why
  - ○ m = n
- → Now, place the value of m to eq-3
- → T(n) = 2^n(T(0)) + 2^(n - 1)k + k
- → T(n) = 2^n*C

- → So, we got time complexity of function is O(2^n).

- → T(n) = 2T(n-2) + k eq—1
- → T(n-2) = 2T(n-4) + k eq—2
- → By putting eq-2 into eq-1
- → T(n) = 2(2T(n-4) + k) + k
- → T(n) = 4T(n-4) + 3k
- → To solve equation until get T (1)
- → We conclude
- → T(n) = 2^m(T(n-2m) + 2^(m-1) k + k eq—3
  - ○ n-2m = 0 (only when 2m will be equal to n)
- → That's why
  - ○ (m = n/2)
- → Now, place the value of m to eq-3
- → T(n) = 2^(n/2) (T(n-2(n/2))) + 2^ (n/2 - 1) k + k
- → T(n) = 2^(n/2) C
- → So, we got time complexity of function is O(2^(n/2)).

- Now we can say that the complexity while using 4 pegs are lesser than time complexity in 3 peg model.
- Also, we saw that the number of movements of disks in 4 peg scenario is comparably lesser that the scenario of 3 peg approach.

3-Peg	**4-Peg**
T1 --> T4	T1 --> T3
T1 --> T3	T1 --> T4
T3 --> T1	T1 --> T2
T1 --> T4	T4 --> T2
T4 --> T3	T3 --> T2
T4 --> T1	T1 --> T3
T1 --> T4	T1 --> T4
T1 --> T3	T3 --> T4
T3 --> T1	T2 --> T1
T3 --> T4	T2 --> T3
T4 --> T3	T2 --> T4
T3 --> T1	T3 --> T4
T1 --> T4	T1 --> T4
T1 --> T3	
T3 --> T1	
T1 --> T4	
T4 --> T3	
T4 --> T1	
T1 --> T4	
T4 --> T3	
T3 --> T1	
T3 --> T4	
T3 --> T1	
T3 --> T1	
T3 --> T1	
T3 --> T4	
T4 --> T3	
T4 --> T1	
T1 --> T4	
T1 --> T3	
T3 --> T1	
T1 --> T4	
T4 --> T3	
T4 --> T1	
T1 --> T4	

Ques-2(a) 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.

Ans-2(a) MT24115_MT24110_Sec1_Q2(a).cpp (Merge sort using recursion)
MT24115_MT24110_Sec1_Q2(b).cpp (Merge sort using Iteration).

Ques-2(b) Explain the iterative code by an example.

Ans-2(b) Lets take an example,
arr[8] = {3,5,6,4,7,9,2,1}

CASE 1:

Size of subarray: i (i = 1 in the first iteration)
where the subarrays will be the size of 2 elements and get sorted by comparing to each other.

[3,5] => [3,5] (No changes already in sorted state)
[6,4] => [4,6]
[7,9] => [7,9] (No changes already in sorted state)
[2,1] => [1,2]

CASE 2:

Size of subarray: i*2 (i = 2 in the second iteration)
Again the same process takes place and the subarray of size 4 get sorted by comparing elements.

[3,5,4,6] => [3,4,5,6]
[7,9,1,2] => [1,2,7,9]

Both the subarrays get sorted now.

CASE 3:

Size of the sub array: i*2^2
Again the exact same process will get executed and the output will be the sorted array for the array of size 8.

[3,4,5,6,1,2,7,9] => [1,2,3,4,5,6,7,9]

The array gets sorted.
Note: The number of cases is dependent on the size of the array.

Ques-3(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’.

Ans-3(a). MT24115_MT24110_Sec1_Q3(a).cpp

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

Ans-3(b). MT24115_MT24110_Sec1_Q3(b).cpp

SECTION – 2

Ques-1(a). You have a custom data structure, DS1, that organizes k items and s upports 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.

Ans-1(a). In this question we have given two conditions to follow
1. Time complexity = k log (k).
2. In-place condition.

Asymptotic Analysis:

Insertion sort: O(k^2*k(log k)) : because of set_at_index(j, x) operations. (K^2 for time complexity of the sorting to compare pair of elements and k(log(k)) for the complexity to swapping of the elememt).

Selection sort: O(k*k(log k)) : because of set_at_index(j, x) operations. (K for time complexity of the sorting to compare pair of elements and k(log(k)) for the complexity to swapping of the elememt).

Merge sort: O(k log k) (since we can use a temporary array to store the merged elements and minimize the number of set_at_index(j, x) operations)

In this scenario we take n size of array. In Insertion sort number of swapping perform at most n^2 and during swap operation then we perform write operation, that's why it takes O(klogk) time per each swapping. So, its overall time complexity is O(n^2*klogk).

Now takes Selection Sort, here we Perform only n or n-1 swapping if given array is randomized. Also, each swapping operation take O(klogk). So here we will get the time complexity O(n^klogk).

Now we perform merge sort, here we are using divide and conquer approach each time we perform O(logn) operation to insert or merging. That's why here time complexity remain same or goes up to O(logn*klogk).

Here we have getting best algorithm is merge sort, but merge sort is not in place algorithm. That's why we here choosing ***Selection Sort*.**

Ques-1(b) Imagine you possess a fixed array A containing references to n cmparable objects, where comparing pairs of objects takes Θ(log k) time. Select the most suitable algorithm to sort the references in A in a manner that ensures the referenced objects appear in non-decreasing order.

Ans-1(b) In this question a complexity (O(logk)) is mentioned to sort pairs of elements and only one out of given three sorting can do that in this complexity.
Insertion and Selection sort does the comparing in O(k) and this complexity is much higher than O(logk) time, that's why we can't choose none of both.
Then we get to choose merge sort.

In insertion and Selection sort time complexity done at O(n^2) and the time complexity for comparing pairs is O(logn) then the total complexity became O(n^2(log(n))) and it will be costly to do that,
instead, we can use merge sort with complexity O(n.log(n)) and the total complexity will be O(n.log(n).log(n))

Ques-1(c). Suppose you are given a sorted array A containing n integers, each of 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.

Ans-1(c). **Asymptotic Analysis:**

Insertion sort: O(n log log n) (since the array is partially sorted and we can take advantage of this)

Selection sort: O(n^2) (since we would need to compare each element with every other element)

Merge sort: O(n log n) (since we would need to merge the entire array, e ven though it's partially sorted)

So the *Insertion sort* is the most suitable algorithm for this scenario because it has the best running time. By taking advantage of the fact that

the array is partially sorted, we can reduce the number of comparisons and insertions needed, resulting in a faster algorithm.

Ques-2(a) 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.)

Ans-2(a). MT24115_MT24110_Sec2_Q2(a1).cpp
MT24115_MT24110_Sec2_Q2(a2).cpp

Ques-2(b). 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.

Ans_2(b). MT24115_MT24110_Sec2_Q2(b).cpp