* Section :- 1

Q1

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

⟹ Section_1_91_a.C code is the implementation of a Variation of the Tower of Hanai classic problem, where here use of two (2) Pegs instead of one (1)

⟹ Work of Solve() function is to transfer disks From source peg to a destination peg using two temporary peg and size of the disk at destination should be in increasing order from bottom to top.

⟹ Solve() function include (2) base case which are test also :- (1). Whe n (number of disks) = 0 then simply return no need of any tranfer required. and (2). When (n = 1) if there is only one disk, it is directly Transfer from source to destination and return.

⟹ Other than base case Solve() function have Recursive case code, where function Firstly recursively moves the top [n-2] disks from the source Peg to tem1 Peg using tem2 Peg and destination Peg.

⇒ Once $(n-2)$ disks Transfer from source to tem1 peg, algorithm starts Transfering $(n-1)^{th}$ disk from source to tem2 peg.

⇒ Next, it move the (n^{th}) largest disk from source to destination Peg.

⇒ After that, it move the $(n-1)^{th}$ disk from tem2 peg to the destination Peg.

⇒ Finally, it recursively move $(n-2)$ disks from tem1 Peg to the destination Peg using source peg and tem2 Peg.

* Pseudo steps to transfer 5 Disks according to the implemented algorithm :-

(1). Transfer 3 disk From T_1 to T_2 using T_3 and T_4 where T_1 = source peg, T_2 = tem1 peg, T_3 = tem2 peg and T_4 = destination Peg.

(2). Transfer disk 4 from T_1 to T_3.

(3) Transfer disk 5 from T_1 to T_4.

(4). Transfer disk 4 from T_3 to T_4.

(5). Transfer 3 disks from T_2 to T_4 using T_1 and T_3

⇒ Let dry run the algorithm and print each steps of Transfering disks from T_1 to T_4.

⟹ Code output :-

Instruction :- Higher the number of disk, higher the size of the disk.

1. Transfer disk 1 from T_1 to T_3
2. Transfer disk 2 from T_1 to T_4
3. Transfer disk 3 from T_1 to T_2
4. Transfer disk 2 from T_4 to T_2
5. Transfer disk 1 from T_3 to T_2
6. Transfer disk 4 from T_1 to T_3
7. Transfer disk 5 from T_1 to T_4
8. Transfer disk 4 from T_3 to T_4
9. Transfer disk 1 from T_2 to T_1
10. Transfer disk 2 from T_2 to T_3
11. Transfer disk 3 from T_2 to T_4
12. Transfer disk 2 from T_3 to T_4
13. Transfer disk 1 from T_1 to T_4

Q1

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

⇒ Analysis Traditional Tower of Hanoi :-

* Algorithm says :-

→ Move (n-1) disks From the source pole to the temporary pole

→ Move (n)th disk from the source pole to the destination pole

→ Move (n-1) disks From the temporary pole to the destination pole.

* Time complexity :-

→ The recurrence relation for traditinal TOH is

$T(n) = 2(T(n-1)) + 1$

$\because T(n-1) = 2(2T(n-2)+1) + 1$

$= 4T(n-2) + 3$

$\because T(n-2) = 4(2T(n-3)+1) + 3$

$= 8T(n-3) + 7$

$\therefore$ In General $T(n) = 2^k T(n-k) + 2^k - 1$

where $n-k=0 \quad \therefore T(0)=0$

$T(n) \approx 2^k - 1 \approx O(2^n)$

* Modified Tower of Hanoi (1 source, 2 temparary, 1 destination pole)

⇒ Algorithm says:-

1. Transfer $(n-2)$ disks from the source pole to the first temparary pole.
2. Move the $(n-1)^{th}$ disk from the source pole to the second temparary pole.
3. Transfer $(n)^{th}$ disk from the source ple to the destination pole.
4. Transfer $(n-1)^{th}$ disk from the second temparary pole to the destination pole.
5. Transfer $(n-2)$ disks from the first temparary pole to the destination pole.

⇒ Time complexity :-

→ The recurrence relation for Modified TOH is

$$T(n) = 2T(n-2) + 3$$

$$\therefore T(n-2) = 2(2T(n-4)+3) + 3$$
$$= 4T(n-4) + 6 + 3$$

$$\therefore T(n-4) = 4(2T(n-6) + 3) + 9$$
$$= 8T(n-6) + 21$$

$$\therefore \text{In General } T(n) = 2^k T(n-2k) + 3(2^{k-1})$$

where $n-2k=0$

$$\therefore T(n) = 0 + 3^{2k} \quad \text{where } k = n/2$$

$$\therefore Tn = 3 \cdot 2^{(n/2)}$$

* Comparison :-

→ Traditional TOH :- $O(2^n)$

→ Modified TOH :- $O(2^{(n/2)})$

⇒ The modified TOH with two temporary poles has a significantly lower time complexity to the traditional TOH.

⇒ So, It seem that the modified TOH algorithm is more efficient and will perform better, especially as the number of disks increases.

Q2

(b) Explain the iterative code by an example.

⇒ Let's take a sample array :- [4, 6, 10, 3, 32, 11].

⇒ On assuming we are having 6 single lits and list containing single element is always sorted.

* Step-by-step code execution :-

(1) First Iteration (P=2) :-

→ Merge subarrays of size 2: by using merge() funtion

→ Merge [4, 6] → [4, 6].

[10, 3] → [3, 10]

[32, 11] → [11, 32]

(2). Second Iteration (P=4) :-

→ Merge subarrays of size 4:

→ Merge [4, 6, 3, 10] → [3, 4, 6, 10]

→ [11, 32] this will not merge in this Iteration

(3) third Iteration (P=8)

→ Merge [4, 6, 3, 10, 11, 32] → [3, 4, 6, 10, 11, 32]

⇒ final sorted array :- [3, 4, 6, 10, 11, 32]

* Section :- 2

Q1

a You have a custom data structure, DS1, that organizes k items and supports two operations :-

(1). DS1.get_at_index(j) takes constant time, and DS1.set_at_index(j, x) takes O(k log k) time.

Your task is to sort the items in DS1 in-place.

⇒ Let's try to analys different scenario for better choices and respective time complexity.

(1). To achieve constant time complexity, we stored data in array so DS1.getat_index(j) satified required time complexity.

(2). To achieve O(klog k) time complexity for DS1.set_at_index(j, x), Let's go through the possible scenario's.

→ Scenario 1: Sorting a small array of integers :-

→ Best choise :- Insertion Sort

→ Insertion sort has a time complexity of $O(n^2)$ in the worst case, but for small array, the constant factors and lower-order terms make it very efficient.

→ It is a In-place sorting algorithm

→ For already partially sorted, It given a best-case time complexity of O(n).

→ But, still on large array size this doesn't suit well.

*→ Scenario 2:- Sorting a large array of Integers

→ Best choice :- Merge sort, because it has a time complexity of $O(n \log n)$ in all the cases.

→ It is a stable sort and work efficient for large array size.

→ But, due to in-place requirement, we can't go with merge sort, because its required extra space while merging subarrays.

*→ Scenario 3:- Sorting array when we have to fullfield In-place requirement.

→ Best choice :- Selection Sort, because it will work in $O(1)$ additional memory requirement.

→ But, due to $O(k \log k)$ time bound, we can't go with Selection sort, because it will take $O(n^2)$ time in all the cases.

*→ Conclusion :-

⇒ Insertion sort :- If we have small array and nearly sorted array due to its simplicity and efficient in these cases.

Q1

b) Imagine you possess a fixed array A containing references of n comparable objects, where comparing pairs of objects takes O(log n) time. Select the most suitable algorithm to sort the reference in A in a manner that ensures the referenced objects appears in non-decreasing order.

→ Let's analysis all 3 given sorting algorithm for best suit the given senario.

① Insertion sort:-

→ Time complexity :- $O(n^2)$

→ Number of comparisons :- $O(n^2)$

→ Not efficient for large arrays because of quadratic time complexity.

② Selection Sort:-

→ Time complexity :- $O(n^2)$

→ Number of comparisons :- $O(n^2)$

→ Not an efficient for large arrays due to quadratic time complexity.

③ Merge sort:-

→ Time complexity :- $O(n \log n)$

→ Number of comparisons :- $O(n \log n)$

→ Best suits for large arrays due to logarithmic number of comparisons.

⇒ So, for given senario comparing 2 objects itself's take log n time, then Best chain is to take merge sort.

⇒ Because it's take only $(n \log n)$ comparisions. and other takes $O(n^2)$.

Q1//

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 pair of adjacent items in A so that A is no long sorted. choose an algorithm to best re-sort the integers in A.

→ As per the question there are in total $\log\log n$ swaps perform, where $\log\log n$ is to small quantity.

→ So, If array is already sorted and $\log\log n$ swaps perform means now array is nearly sorted.

→ An algorith wd which suit best in case of nearly sorted array is Insertion sort.

→ Insertion sort is particularly efficient for nearly sorted arrays, with a best-time complexity of $O(n)$.

→ Given that only $\log\log n$ swaps were performed, the array is very close to being sorted, making Insertion sort an ideal choice.

→ The average and worst case time complexity of Insertion sort is $O(n^2)$

→ But, for nearly sorted arrays, it performs much better in practice.