Date ___/___/______

Set 3

1. Explain the concept of algorithm efficiency and how it is measured. 3M

⇒ Algorithm efficiency means how well an algorithm use computer resources like time and memory to solve a problem.

⇒ It tells us how fast and how much ~~tim~~ size /space an algorithm needs to give the output.

→ There are two main factors to measure algorithm efficiency.

1.) Time efficiency (Time complexity):

⇒ Measures howmuch time an algorithm takes to execute output.

common TCs = $O(n)$, $O(n^2)$, $\log(n)$, $O(1)$;

2.) Space efficiency (Space complexity):

⇒ Measures how much memory the algorithm uses while executing.

Common TCs = $O(1)$, $O(n)$, $O(n^2)$, $\log(n)$;

2. Differentiate between time complexity and space complexity with examples. 3M

Basis	Time Complexity	Space Complexity
Def	It measures how much time an algo takes to execute.	Space
Focus on	speed or execution time	Memory storag req.
Depends on	Number of operations or steps executed	Number of variables data st., Rec. depth.
Goal	To make algo run faster.	To make the algo use less memory.
Examples:	① `for(i=0; i<n; i++){ cout<<n<<" "; }`	
⇒	loop runs n times so TC = O(n)	use only constant var so; SC = O(1)
	`int fact(int n){ if (n == 1) return 1; return n* fact(n-1); }`	
⇒	calls itself n times TC = O(n) time	Each recursive call stores in stack SC = O(n) space

Date __/__/____

3. Derive the index formula for accessing elements in a 3-D array in column-major order. 4 M

Address of

$$A[I][J][K] = B + W*(N*L*(I-x) + (J-y) + (K-z)*N)$$

B = Base Add
W = Size of elem
N = Height / Layer
M = Row (total no of rows)
L = Col (" " " columns)
x, y, z = Lower bound of Row, Col, Height.

4.) Write the algo for binary search and explain its steps. 4 M

⇒ Set 1 Q. no. 4.

5. Describe bubble sort and explain its working with example. 5 M

⇒ Bubble sort is a sorting algorithm where we repeatedly go through the list, compare two neighboring numbers, and swap them if they are in wrong order until the list is sorted.

Algorithm: 1. Start from the first element and compare it with the next element.
2. If the first element is greater than the next one, swap them.
3. Move to the next pair of elements and repeat the comparision and swapping.
4. Continue this process until the largest elem "bubbles up" to the end of the list.

SM

Date __/__/__

5. then repean the same prozess for the remaining elems.
6. Keep repeating untill no swaps needed.

Example!

arr = [5, 3, 8, 4, 2]

Pass 1!

compare (5, 3) → swap [3, 5, 8, 4, 2]
compare (5, 8) → no swap [3, 5, 8, 4, 2]
compare (8, 4) → swap [3, 5, 4, 8, 2]
compare (8, 2) → swap [3, 5, 4, 2, 8]
" after pass 1 Largest elem is at end."

pass 2!

compare (3, 5) → No Swap
compare (5, 4) swap [3, 4, 5, 2, 8]
compare (5, 2) swap [3, 4, 2, 5, 8]
" second largest elem at its correct pos"

pass 3!

compare (3, 4) → no swap
compare (4, 2) → swap [3, 2, 4, 5, 8]

pass 3!

compare (3, 2) → swap [2, 3, 4, 5, 8]
Array is sorted

```
for (int i = 0; i < n-1; i++) {
    for (int j = 0; j < n-1-i; j++) {
        if (arr[j] < arr[j+1]) {
            swap(arr[j], arr[j+1]) }
} }
```

6. Write a recursive method for calculating factorial of a number and explain it. 5 M

⇒ We know that factorial is a recursive method

factorial of n = n* factorial of (n-1);
we already know factorial of 1 is 1.

Algorithm :

i) Start
ii) Read the number
iii) Define recursive function with base case.

```
if n = 0 or n = 1
    Return 1;
Else
    Return n x ~~n-1~~ factorial(n-1);
```

iv) call the function.
v) Display result end.

SM

```
int fact(int n) {
    if (n==0 || n==1) {
        return 1;
    }
    return n* fact(n-1);
}
```

- fact(4)
- ⇒ 4 x fact (3)
- ⇒ 4 x 3 x (fact (2))
- ⇒ 4 x 3 x 2 x fact (1) // reached base case.
- ⇒ 4 x 3 x 2 x 1
- ⇒ 24.

Recursion tree.

fact(4) = 24

4 x fact(3) = 24

3 x fact(2) = 6

2 x fact(1) = 2

1

$\boxed{\text{So fact (4) = 24}}$

Date __/__/____

7 Explain circular linked Lists and write the algorithm to traverse it. 8 M

⇒ A circular linked list is a type of Linked List in which the last node points back to the first node, instead of Null.

Types:

1. Singly circular Linked list
→ Each node has one pointer (next).
→ The next of the last node points to the head node.

2. Doubly circular Linked List
→ Each node has two pointer (next and prev).
→ The last node's next pointer points to the head and head's prev points to the last node.

Algorithm to traverse it.

1. Start
2. Initialize a pointer PTR and PTR = Head.
3. if Head = Null then
   print "List is empty".
   Go to 7
4. Print the data of the node pointed by PTR.
5. Move PTR to the next node
   PTR = PTR → next.
6. Repeat untill PTR again equal to Head.
7. Stop.

Date ___/___/______

$\boxed{\text{code}}$

```
void traverse (Node* Head){
    if (head == NULL){
        cout<<"List Empty";
        return;
    }

    Node* temp = Head ;

    do {
        cout<< temp->data<<" ";
        temp = temp -> next;
    }
    while(temp != head);
}
```

8 (a) Discuss the trade-offs b/w recursion and iteration with reference to memory usage. 4M

⇒ Recursion!

→ In recursion, each function call is stored in call stack.

→ Every recursive call keeps its own copy of

→ Local vars

→ Parameter

→ Return address.

Date __/__/____

Therefore, recursion uses extra stack memory for each call.

e.g factorial of a number

for fact(5), the stack stores function calls → fact(5), fact(4) --- fact(1)

⇒ Memory Usage:
→ Grows linearly with the depth of rec → O(n) space
→ can cause stack overflow if recursion is too deep.

⇒ Iteration:
→ Iteration uses loop and single set of vars.
→ No stack frame is created - the same memory is reused in each iteration.

e.g factorial of a number

⇒ Memory Usage:
→ Only a few vars are used like (i, result) → O(1) space
→ No risk of stack overflow.

Date __/__/____

8 (b) Write a recursive algorithm for the fibbonacci series. 4 marks.

fibbonacc series is defined as

$$F(0) = 0, \; F(1) = 1$$
$$F(n) = F(n-1) + F(n-2).$$

Algorithm

→ Start
→ if N = 0 or N = 1 return 0, 1.
→ else call function fun(n-1) + func(n-2);.
→ return fun(n-1) + fun(n-2);.
→ end.

$\boxed{\text{Code}}$

```
int fib(int n) {

    if(n==0) { return 0;}

    else if (n==1){ return 1; }
    else{
        return fib(n-1) +fib(n-2);
    }
}
```

Date ___/___/_____

9. Given unsorted array, explain how merge sort sorts it efficiently. Also discuss realworld applications where merge sort is prefered. 10 M

⇒ Merge Sort → Set 1 Q.9

Merge sort real world applications:

1) External sorting (Big data)
→ Used when data is too large to fit in memory.

ii) Database
→ Sorting records efficiently before merging/ joining tables

iii) File System
→ Used to merge and sort large files.

iv) Financial System
→ Used in stock trading plateforms and banking systems where large transaction records must be sorted by time, amount, or account efficiency.

Date ___/___/______

9 (OR) Describe the implementation and application of singly Linked List in managing polynomial expressions. 10M

⇒ Implementation :

⇒ Algo: (create polynomial)

→ Start

→ Initialize Head = Null

→ For each term (coefficient, exponent):

a : Create a new node

b : Assign New → coeff = coefficient,
New → pow = exponent

c : If Head = null then
Set Head = New
Set New → next = null.
else
Traverse to the end of the list
Attach New to the last node

→ End.

Code

```
// Function to create node

Node* createNode(int coeff, int pow) {
    Node* newNode = new Node();
    newNode -> coeff = coeff;
    newNode -> pow = pow;
    newNode -> next = NULL;
    return newNode;
}
```

Date __/__/____

// Function to insert term at the end of list.

```
void insertTerm(Node*& head, int coeff, int pow){
Node* newNode = createNode(coeff, pow);
  if (head == nullptr){
      head = newNode;
     return;
  }

  Node* temp = head;
  While (temp->next != nullptr){
      temp = temp -> next;
  }
  temp -> next = newNode;
}
```

Application:

⇒ A polynomial expression is mathematical expression that contains variables and coefficients, like:

$$P(x) = 5x^3 + 4x^2 + 2x + 1$$

⇒ Managing polynomial using arrays is inefficient when inserting, deleting, or adding new terms because array size is fixed.

⇒ To overcome this, we use singly LL, where each node stores one term of the polynomial.