**Part A: Algorithm Writing & Analysis**

1. Write algorithms (pseudo-code) for the following problems:
2. Find the maximum and minimum element in a list of n integers.
   1. array[1.n]
   2. max = array[0]
   3. min = array[0]
   4. For i=1 to n , i++
      1. If array[i] > max ;
         1. max = array[i]
      2. If array[i] < min;
         1. min = array[i]
   5. return max , min
3. Count Number of Odd and Even Numbers in an Array
   1. Array <- A[n]
   2. Odd <- 0
   3. even <- 0
   4. for i=1 to n ,i++
      1. if A[i] mod(%) 2 = 0 then
         1. even += 1
      2. else
         1. odd += 1
   5. return even , odd
4. Reverse a Given Array
   * 1. Algorithm ReverseArray(A[1.n])
     2. Input: Array A of n integers
     3. Output: Reversed array A

1 left ← 1

2. right ← n

3. while left < right do

4. temp ← A[left]

5. A[left] ← A[right]

6. A[right] ← temp

7. left ← left + 1

8. right ← right - 1

9. return A

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Time Complexity (Big O notation )** | **Space Complexity** | **Optimality** |
| Find Max and Min | O(n) | O(1) | Optimal (single pass) |
| Count Odd and Even | O(n) | O(1) | Optimal (single pass) |
| Reverse Array | O(n) | O(1) | Optimal (in-place reversal) |

Part B : Asymptotic Analysis

• Write the asymptotic analysis neatly for each snippet.

• Mention:

o Best-case, Worst-case, and Average-case complexities (where applicable).

o The total number of operations step by step.

3. Given the following code snippet, determine its time complexity and justify your

#answer:

A)

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

printf(" \* ");

}

}

Operations:

Outer loop runs **n** times.

Inner loop runs **n** times for each iteration of the outer loop.

Total operations = n × n = n².

|  |  |  |
| --- | --- | --- |
| Time Complexity | | |
| Best-case | Worst-case | Average-case |
| O(n²) | O(n²) | O(n²) |

B)

void func(int n) {

if (n == 1)

return;

func(n/2);

func(n/2);

}

Operations:

This is a recursive function that calls itself twice with input n/2.

It forms a binary recursion tree with log(n) depth and 2^log(n) = n leaves.

|  |  |  |
| --- | --- | --- |
| Time Complexity | | |
| Best-case | Worst-case | Average-case |
| O(n) | O(n) | O(n) |

C)

int i = 1;

while (i < n) {

printf("%d ", i);

i = i \* 2;

}

Operations:

i grows exponentially (1, 2, 4, 8, ...) until it reaches n.

So total operations = log₂(n)

|  |  |  |
| --- | --- | --- |
| Time Complexity | | |
| Best-case | Worst-case | Average-case |
| O(log n) | O(log n) | O(log n) |

D)

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

for (int k = 0; k < n; k++) {

printf("\*");

}

}

}

Operations:

3 nested loops, each running n times.

Total = n × n × n = n³

|  |  |  |
| --- | --- | --- |
| Time Complexity | | |
| Best-case | Worst-case | Average-case |
| O(n³) | O(n³) | O(n³) |

D)

int fib(int n) {

if (n <= 1)

return n;

return fib(n - 1) + fib(n - 2);

}

Operations:

Exponential recursive calls due to overlapping subproblems.

Forms a binary tree with ~2^n nodes.

|  |  |  |
| --- | --- | --- |
| Time Complexity | | |
| Best-case | Worst-case | Average-case |
| O(2ⁿ) | O(2ⁿ) | O(2ⁿ) |

F)

printf("Hello World");

Operations: Just a single print statement.

|  |  |  |
| --- | --- | --- |
| Time Complexity | | |
| Best-case | Worst-case | Average-case |
| O(1) | O(1) | O(1) |

G)

for (int i = 0; i < n; i++) {

printf("\*");

}

for (int j = 0; j < n; j++) {

for (int k = 0; k < n; k++) {

printf("#");

}

}

Operations:

First loop: n times → O(n)

Second loop: nested n × n → O(n²)

Total = O(n + n²) = O(n²)

|  |  |  |
| --- | --- | --- |
| Time Complexity | | |
| Best-case | Worst-case | Average-case |
| O(n²) | O(n²) | O(n²) |