

Time And Space Complexity MCQ

Time & Space Complexity Practice Problems

A comprehensive guide with 30 problems ranging from basic to hard level. Each problem includes code in C/C++ and multiple choice options to test your understanding.

BASICS (Problems 1-8)

Problem 1: Simple Assignment

```
int x = 5;  
int y = 10;  
int z = x + y;
```

What is the Time Complexity?

- A) $O(1)$
- B) $O(n)$
- C) $O(\log n)$
- D) $O(n^2)$

What is the Space Complexity?

- A) $O(1)$
- B) $O(n)$
- C) $O(\log n)$
- D) $O(n^2)$

ANSWER: Time Complexity: A) $O(1)$ Space Complexity: A) $O(1)$

Explanation: This code performs a fixed number of operations (3 assignments) regardless of input size. It uses constant space (3 integer variables). No loops, no recursion, no input-dependent operations.

Problem 2: Single Loop

```
void printNumbers(int n) {  
    for(int i = 0; i < n; i++) {  
        cout << i << " ";  
    }  
}
```

What is the Time Complexity?

- A) $O(1)$
- B) $O(n)$
- C) $O(\log n)$
- D) $O(n^2)$

What is the Space Complexity?

- A) $O(1)$
- B) $O(n)$
- C) $O(\log n)$
- D) $O(n^2)$

ANSWER: Time Complexity: B) $O(n)$ Space Complexity: A) $O(1)$

Explanation: The loop runs exactly n times, so time complexity is $O(n)$. Space complexity is $O(1)$ because we only use a single variable `i` regardless of n 's value. The function doesn't allocate any additional memory proportional to n .

Problem 3: Two Sequential Loops

```
void twoLoops(int n) {
    for(int i = 0; i < n; i++) {
        printf("%d ", i);
    }

    for(int j = 0; j < n; j++) {
        printf("%d ", j);
    }
}
```

What is the Time Complexity?

- A) $O(n)$
- B) $O(2n)$
- C) $O(n^2)$
- D) $O(\log n)$

What is the Space Complexity?

- A) $O(1)$
- B) $O(n)$
- C) $O(2n)$
- D) $O(n^2)$

ANSWER: Time Complexity: A) $O(n)$ Space Complexity: A) $O(1)$

Explanation: First loop runs n times, second loop runs n times = $n + n = 2n$ operations. In Big O notation, we drop constants, so $O(2n) = O(n)$. Space is constant since we only use two loop variables.

Problem 4: Array Access

```
int getElement(int arr[], int n, int index) {
    return arr[index];
}
```

What is the Time Complexity?

- A) $O(1)$
- B) $O(n)$
- C) $O(\log n)$
- D) $O(n^2)$

What is the Space Complexity?

- A) $O(1)$
- B) $O(n)$
- C) $O(\log n)$
- D) $O(n^2)$

ANSWER: Time Complexity: A) $O(1)$ Space Complexity: A) $O(1)$

Explanation: Array access by index is a direct memory access operation that takes constant time regardless of array size. We don't allocate any new memory in this function, so space is $O(1)$. Note: The array itself is passed by reference, so we don't count it in space complexity.

Problem 5: Loop with Division

```
void divideLoop(int n) {  
    for(int i = n; i > 0; i = i/2) {  
        printf("%d ", i);  
    }  
}
```

What is the Time Complexity?

- A) $O(n)$
- B) $O(\log n)$
- C) $O(n \log n)$
- D) $O(n^2)$

What is the Space Complexity?

- A) $O(1)$
- B) $O(n)$
- C) $O(\log n)$
- D) $O(n^2)$

ANSWER: Time Complexity: B) $O(\log n)$ Space Complexity: A) $O(1)$

Explanation: The loop divides i by 2 in each iteration. Starting from n : n , $n/2$, $n/4$, $n/8$... until it reaches 0. The number of iterations is $\log_2(n)$. This is logarithmic time. Space is constant as we only use variable i .

Problem 6: Linear Search

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

What is the Time Complexity (Worst Case)?

- A) $O(1)$
- B) $O(n)$
- C) $O(\log n)$
- D) $O(n^2)$

What is the Space Complexity?

- A) $O(1)$
- B) $O(n)$
- C) $O(\log n)$

- D) $O(n^2)$

ANSWER: Time Complexity: B) $O(n)$ Space Complexity: A) $O(1)$

Explanation: In the worst case (element not found or at the end), we check all n elements. Best case is $O(1)$ if found at first position, but we analyze worst case. Space is $O(1)$ because we only use variable i and don't allocate additional memory.

Problem 7: Sum of Array

```
int sumArray(int arr[], int n) {  
    int sum = 0;  
    for(int i = 0; i < n; i++) {  
        sum += arr[i];  
    }  
    return sum;  
}
```

What is the Time Complexity?

- A) $O(1)$
- B) $O(n)$
- C) $O(\log n)$
- D) $O(n^2)$

What is the Space Complexity?

- A) $O(1)$
- B) $O(n)$
- C) $O(\log n)$
- D) $O(n^2)$

ANSWER: Time Complexity: B) $O(n)$ Space Complexity: A) $O(1)$

Explanation: We iterate through the array once, visiting each of n elements exactly once, giving $O(n)$ time. We only use two extra variables (sum and i),

regardless of array size, so space is $O(1)$.

Problem 8: Loop with Multiplication

```
void multiplyLoop(int n) {
    for(int i = 1; i < n; i = i * 2) {
        cout << i << " ";
    }
}
```

What is the Time Complexity?

- A) $O(n)$
- B) $O(\log n)$
- C) $O(n \log n)$
- D) $O(n^2)$

What is the Space Complexity?

- A) $O(1)$
- B) $O(n)$
- C) $O(\log n)$
- D) $O(n^2)$

ANSWER: Time Complexity: B) $O(\log n)$ Space Complexity: A) $O(1)$

Explanation: Loop iterations: 1, 2, 4, 8, 16... (powers of 2) until reaching n. Number of iterations is $\log_2(n)$. Similar to binary search pattern. Space is constant with only variable i.

EASY (Problems 9-15)

Problem 9: Nested Loop (Square)

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

```
    for(int j = 0; j < n; j++) {  
        printf("%d,%d ", i, j);  
    }  
}  
}
```

What is the Time Complexity?

- A) $O(n)$
- B) $O(\log n)$
- C) $O(n^2)$
- D) $O(2n)$

What is the Space Complexity?

- A) $O(1)$
- B) $O(n)$
- C) $O(n^2)$
- D) $O(\log n)$

ANSWER: Time Complexity: C) $O(n^2)$ Space Complexity: A) $O(1)$

Explanation: Outer loop runs n times. For each iteration of outer loop, inner loop also runs n times. Total operations = $n \times n = n^2$. This is quadratic time complexity. Space is constant as we only use i and j variables.

Problem 10: Array Copy

```
void copyArray(int source[], int dest[], int n) {  
    for(int i = 0; i < n; i++) {  
        dest[i] = source[i];  
    }  
}
```

What is the Time Complexity?

- A) $O(1)$

- B) $O(n)$
- C) $O(\log n)$
- D) $O(n^2)$

What is the Space Complexity (excluding input arrays)?

- A) $O(1)$
- B) $O(n)$
- C) $O(\log n)$
- D) $O(n^2)$

ANSWER: Time Complexity: B) $O(n)$ Space Complexity: A) $O(1)$

Explanation: Single loop copying n elements = $O(n)$ time. For space complexity, we typically analyze auxiliary space (extra space used by the algorithm). The destination array is part of the output, not auxiliary space. We only use variable i as extra space = $O(1)$.

Problem 11: Triangle Pattern Loop

```
void trianglePattern(int n) {
    for(int i = 0; i < n; i++) {
        for(int j = 0; j <= i; j++) {
            printf("* ");
        }
        printf("\n");
    }
}
```

What is the Time Complexity?

- A) $O(n)$
- B) $O(n^2)$
- C) $O(n \log n)$
- D) $O(2n)$

What is the Space Complexity?

- A) $O(1)$
- B) $O(n)$
- C) $O(n^2)$
- D) $O(\log n)$

ANSWER: Time Complexity: B) $O(n^2)$ Space Complexity: A) $O(1)$

Explanation: Inner loop runs: 1, 2, 3, ... n times. Total = $1+2+3+\dots+n = n(n+1)/2 = (n^2+n)/2$. Dropping constants and lower order terms: $O(n^2)$. Space is constant with only i and j variables.

Problem 12: Binary Search

```
int binarySearch(int arr[], int left, int right, int target) {
    while(left <= right) {
        int mid = left + (right - left) / 2;

        if(arr[mid] == target)
            return mid;

        if(arr[mid] < target)
            left = mid + 1;
        else
            right = mid - 1;
    }
    return -1;
}
```

What is the Time Complexity?

- A) $O(n)$
- B) $O(\log n)$
- C) $O(n \log n)$

- D) $O(n^2)$

What is the Space Complexity?

- A) $O(1)$
- B) $O(n)$
- C) $O(\log n)$
- D) $O(n^2)$

ANSWER: Time Complexity: B) $O(\log n)$ Space Complexity: A) $O(1)$

Explanation: Binary search divides the search space in half each iteration. Starting with n elements: $n, n/2, n/4, n/8...$ until 1. Number of divisions = $\log_2(n)$. This is iterative binary search using constant extra space (left, right, mid variables only).

Problem 13: Reverse Array

```
void reverseArray(int arr[], int n) {  
    int start = 0;  
    int end = n - 1;  
  
    while(start < end) {  
        int temp = arr[start];  
        arr[start] = arr[end];  
        arr[end] = temp;  
        start++;  
        end--;  
    }  
}
```

What is the Time Complexity?

- A) $O(1)$
- B) $O(n)$
- C) $O(\log n)$

- D) $O(n^2)$

What is the Space Complexity?

- A) $O(1)$
- B) $O(n)$
- C) $O(\log n)$
- D) $O(n^2)$

ANSWER: Time Complexity: B) $O(n)$ Space Complexity: A) $O(1)$

Explanation: The loop runs $n/2$ times (meeting in the middle). $O(n/2) = O(n)$ after dropping constants. This is in-place reversal using only three variables (start, end, temp), so space is $O(1)$.

Problem 14: Find Maximum

```
int findMax(int arr[], int n) {  
    int max = arr[0];  
    for(int i = 1; i < n; i++) {  
        if(arr[i] > max) {  
            max = arr[i];  
        }  
    }  
    return max;  
}
```

What is the Time Complexity?

- A) $O(1)$
- B) $O(n)$
- C) $O(\log n)$
- D) $O(n^2)$

What is the Space Complexity?

- A) $O(1)$

- B) $O(n)$
- C) $O(\log n)$
- D) $O(n^2)$

ANSWER: Time Complexity: B) $O(n)$ Space Complexity: A) $O(1)$

Explanation: We must check all n elements to find the maximum, so time is $O(n)$. We only use two variables (max and i) regardless of array size, giving $O(1)$ space. Cannot be optimized to less than $O(n)$ time for unsorted array.

Problem 15: Nested Loop with Break

```
void searchMatrix(int arr[], int n, int target) {
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            if(arr[i] == target) {
                printf("Found\n");
                break;
            }
        }
    }
}
```

What is the Time Complexity (Worst Case)?

- A) $O(n)$
- B) $O(\log n)$
- C) $O(n^2)$
- D) $O(n \log n)$

What is the Space Complexity?

- A) $O(1)$
- B) $O(n)$
- C) $O(n^2)$

- D) $O(\log n)$

ANSWER: Time Complexity: C) $O(n^2)$ Space Complexity: A) $O(1)$

Explanation: Break only exits the inner loop, not the outer loop. Worst case: target never found, so outer loop runs n times and inner loop runs n times for each = n^2 . Best case would be $O(1)$ if found immediately. Space is constant.

MEDIUM (Problems 16-23)

Problem 16: Bubble Sort

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

What is the Time Complexity (Worst Case)?

- A) $O(n)$
- B) $O(n \log n)$
- C) $O(n^2)$
- D) $O(2^n)$

What is the Space Complexity?

- A) $O(1)$
- B) $O(n)$
- C) $O(\log n)$

- D) $O(n^2)$

ANSWER: Time Complexity: C) $O(n^2)$ Space Complexity: A) $O(1)$

Explanation: Outer loop runs $n-1$ times. Inner loop runs: $(n-1)$, $(n-2)$, ... 1 times. Total = $(n-1) + (n-2) + \dots + 1 = n(n-1)/2 \approx n^2/2 = O(n^2)$. Bubble sort is in-place, using only constant extra space (temp, i, j).

Problem 17: Recursive Factorial

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

What is the Time Complexity?

- A) $O(1)$
- B) $O(n)$
- C) $O(\log n)$
- D) $O(n^2)$

What is the Space Complexity?

- A) $O(1)$
- B) $O(n)$
- C) $O(\log n)$
- D) $O(n^2)$

ANSWER: Time Complexity: B) $O(n)$ Space Complexity: B) $O(n)$

Explanation: Function calls itself n times: $\text{factorial}(n) \rightarrow \text{factorial}(n-1) \rightarrow \dots \rightarrow \text{factorial}(1)$. Each call does constant work, so time is $O(n)$. Space is $O(n)$ because of the call stack. Each recursive call adds a frame to the stack, and we have n frames at maximum depth.

Problem 18: Fibonacci (Recursive - Naive)

```
int fibonacci(int n) {  
    if(n <= 1) {  
        return n;  
    }  
    return fibonacci(n-1) + fibonacci(n-2);  
}
```

What is the Time Complexity?

- A) $O(n)$
- B) $O(\log n)$
- C) $O(n^2)$
- D) $O(2^n)$

What is the Space Complexity (Call Stack)?

- A) $O(1)$
- B) $O(n)$
- C) $O(\log n)$
- D) $O(2^n)$

ANSWER: Time Complexity: D) $O(2^n)$ Space Complexity: B) $O(n)$

Explanation: Each call makes 2 recursive calls, creating a binary tree of calls. Tree has 2^n nodes approximately = $O(2^n)$ time. This is extremely inefficient! Space is $O(n)$ for call stack depth (not $O(2^n)$) because at any moment, only one path from root to leaf exists in memory, and maximum depth is n .

Problem 19: Two Pointer - Pair Sum

```
bool findPairWithSum(int arr[], int n, int target) {  
    int left = 0;  
    int right = n - 1;
```

```

while(left < right) {
    int sum = arr[left] + arr[right];
    if(sum == target) {
        return true;
    } else if(sum < target) {
        left++;
    } else {
        right--;
    }
}
return false;
}

```

What is the Time Complexity (assuming sorted array)?

- A) $O(n)$
- B) $O(n^2)$
- C) $O(\log n)$
- D) $O(n \log n)$

What is the Space Complexity?

- A) $O(1)$
- B) $O(n)$
- C) $O(\log n)$
- D) $O(n^2)$

ANSWER: Time Complexity: A) $O(n)$ Space Complexity: A) $O(1)$

Explanation: Two pointers move toward each other, each moving at most n positions total. In worst case, we examine all n elements once = $O(n)$. This is much better than nested loop $O(n^2)$ approach! Space is $O(1)$ using only three variables.

Problem 20: Merge Sort

```

void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int L[n1], R[n2];

    for(int i = 0; i < n1; i++) L[i] = arr[left + i];
    for(int j = 0; j < n2; j++) R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;
    while(i < n1 && j < n2) {
        if(L[i] <= R[j]) {
            arr[k++] = L[i++];
        } else {
            arr[k++] = R[j++];
        }
    }
    while(i < n1) arr[k++] = L[i++];
    while(j < n2) arr[k++] = R[j++];
}

void mergeSort(int arr[], int left, int right) {
    if(left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

```

What is the Time Complexity?

- A) $O(n)$
- B) $O(n \log n)$
- C) $O(n^2)$

- D) $O(\log n)$

What is the Space Complexity?

- A) $O(1)$
- B) $O(n)$
- C) $O(\log n)$
- D) $O(n^2)$

ANSWER: Time Complexity: B) $O(n \log n)$ Space Complexity: B) $O(n)$

Explanation: Array is divided $\log(n)$ times (binary division). At each level, we merge n elements = $O(n)$ work per level. Total = $O(n \log n)$. Space is $O(n)$ for temporary arrays in merge operations plus $O(\log n)$ for call stack $\approx O(n)$ dominant term.

Problem 21: Quick Sort (Partition)

```
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for(int j = low; j < high; j++) {
        if(arr[j] < pivot) {
            i++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
    return i + 1;
}

void quickSort(int arr[], int low, int high) {
```

```

    if(low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

```

What is the Time Complexity (Average Case)?

- A) $O(n)$
- B) $O(n \log n)$
- C) $O(n^2)$
- D) $O(\log n)$

What is the Space Complexity (Average Case)?

- A) $O(1)$
- B) $O(n)$
- C) $O(\log n)$
- D) $O(n^2)$

ANSWER: Time Complexity: B) $O(n \log n)$ [Average], C) $O(n^2)$ [Worst] Space Complexity: C) $O(\log n)$ [Average]

Explanation: Average case: Partition divides array roughly in half, leading to $\log(n)$ depth with $O(n)$ work per level = $O(n \log n)$. Worst case (already sorted): $O(n^2)$. Space is $O(\log n)$ for recursion stack in average case. Quick sort is in-place but uses stack space.

Problem 22: String Permutation Check

```

bool arePermutations(string str1, string str2) {
    if(str1.length() != str2.length()) {
        return false;
    }
}

```

```

int count[256] = {0};

for(int i = 0; i < str1.length(); i++) {
    count[str1[i]]++;
    count[str2[i]--];
}

for(int i = 0; i < 256; i++) {
    if(count[i] != 0) {
        return false;
    }
}
return true;
}

```

What is the Time Complexity?

- A) $O(1)$
- B) $O(n)$
- C) $O(n^2)$
- D) $O(n \log n)$

What is the Space Complexity?

- A) $O(1)$
- B) $O(n)$
- C) $O(\log n)$
- D) $O(n^2)$

ANSWER: Time Complexity: B) $O(n)$ Space Complexity: A) $O(1)$

Explanation: First loop runs n times (string length), second loop runs 256 times (constant). $O(n + 256) = O(n)$. The count array is always size 256 regardless of input size, so space is $O(1)$ constant.

Problem 23: Matrix Multiplication

```

void multiplyMatrices(int A[][100], int B[][100], int C[][100], int n) {
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            C[i][j] = 0;
            for(int k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

```

What is the Time Complexity?

- A) $O(n)$
- B) $O(n^2)$
- C) $O(n^3)$
- D) $O(n \log n)$

What is the Space Complexity (excluding input/output matrices)?

- A) $O(1)$
- B) $O(n)$
- C) $O(n^2)$
- D) $O(n^3)$

ANSWER: Time Complexity: C) $O(n^3)$ Space Complexity: A) $O(1)$

Explanation: Three nested loops, each running n times = $n \times n \times n = n^3$. This is cubic time complexity. For auxiliary space (not counting input/output matrices), we only use three loop variables $i, j, k = O(1)$ constant space.

HARD (Problems 24-26)

Problem 24: Tower of Hanoi

```

void towerOfHanoi(int n, char from, char to, char aux) {
    if(n == 1) {
        cout << "Move disk 1 from " << from << " to " << to << endl;
        return;
    }
    towerOfHanoi(n-1, from, aux, to);
    cout << "Move disk " << n << " from " << from << " to " << to << endl;
    towerOfHanoi(n-1, aux, to, from);
}

```

What is the Time Complexity?

- A) $O(n)$
- B) $O(n \log n)$
- C) $O(n^2)$
- D) $O(2^n)$

What is the Space Complexity?

- A) $O(1)$
- B) $O(n)$
- C) $O(\log n)$
- D) $O(2^n)$

ANSWER: Time Complexity: D) $O(2^n)$ Space Complexity: B) $O(n)$

Explanation: Recurrence: $T(n) = 2T(n-1) + 1$. Solving: $T(n) = 2^n - 1 = O(2^n)$. Each call makes 2 recursive calls. Total moves = $2^n - 1$. Space is $O(n)$ for call stack depth (maximum n recursive calls on stack at once). This grows exponentially!

Problem 25: All Subsets (Power Set)

```

void printSubsets(int arr[], int n) {
    int totalSubsets = 1 << n; //  $2^n$ 

```

```

for(int i = 0; i < totalSubsets; i++) {
    printf("{ ");
    for(int j = 0; j < n; j++) {
        if(i & (1 << j)) {
            printf("%d ", arr[j]);
        }
    }
    printf("}\n");
}
}

```

What is the Time Complexity?

- A) $O(n)$
- B) $O(n^2)$
- C) $O(n * 2^n)$
- D) $O(2^n)$

What is the Space Complexity?

- A) $O(1)$
- B) $O(n)$
- C) $O(2^n)$
- D) $O(n * 2^n)$

ANSWER: Time Complexity: C) $O(n * 2^n)$ Space Complexity: A) $O(1)$

Explanation: Outer loop runs 2^n times (all possible subsets). Inner loop runs n times for each subset. Total = $2^n \times n = O(n * 2^n)$. Space is $O(1)$ as we only use loop variables i and j , not storing all subsets in memory.

Problem 26: Longest Common Subsequence (Dynamic Programming)

```

int lcs(string s1, string s2, int m, int n) {
    int dp[m+1][n+1];

```

```

for(int i = 0; i <= m; i++) {
    for(int j = 0; j <= n; j++) {
        if(i == 0 || j == 0) {
            dp[i][j] = 0;
        }
        else if(s1[i-1] == s2[j-1]) {
            dp[i][j] = dp[i-1][j-1] + 1;
        }
        else {
            dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
        }
    }
}
return dp[m][n];
}

```

What is the Time Complexity?

- A) $O(m + n)$
- B) $O(m * n)$
- C) $O(m^2 * n^2)$
- D) $O(2^{(m+n)})$

What is the Space Complexity?

- A) $O(1)$
- B) $O(m + n)$
- C) $O(m * n)$
- D) $O(m^2 * n^2)$

ANSWER: Time Complexity: B) $O(m * n)$ Space Complexity: C) $O(m * n)$

Explanation: Two nested loops: outer runs $m+1$ times, inner runs $n+1$ times = $(m+1)(n+1) \approx mn = O(mn)$. The dp table is of size $(m+1) \times (n+1) = O(m*n)$ space.

This is the classic DP solution trading space for time efficiency (much better than exponential recursive solution).
