**Activities**

**Number 1**

1. The time complexity is O(n^2), because the time it takes for the outer loop is multiplied by the inner loop which both takes n times. So, n \* n = n^2
2. The time complexity is O(n), because the outer loop takes O(n) time and the inner loop time is constant time O(1).
3. The time complexity is O(n^2), because the time it takes for the outer loop is multiplied by the inner loop which each takes n times and n-1 respectively. So, both loops depend on n.
4. The time complexity is O(n^2), because the time it takes for the outer loop is multiplied by the inner loop which each takes n-1 times and n respectively. So, both loops depend on n.

**Number 2**

1. For the first loop: [0,1,2,3,3,3,3,3]
2. For the second loop: [0,1,2,3,3,4,5,6]

**Number 3**

1. Sum of an array:

* T(n) = O(1) + O(n) + O(1)
* O(n) = O(n)

1. Matrix multiplication:

* T(n) = O(r1×c2×c1)
* O(n) = O(r1xc2)

1. For looping:

* T(n) = O(n)
* O(n) = O(1)

1. While looping:

* T(n) = O(log n)
* O(n) = O(1)

**Number 4**

1. Accessing an element by index in an array

Time complexity: O(1) - Constant Time Complexity

1. Binary Search

Time complexity: O(log n) - Logarithmic Time Complexity

1. Linear Search

Time complexity: O(n) - Linear Time Complexity

1. Merge Sort, Heap Sort

Time complexity: O(n log n) - Linearithmic Time Complexity

1. Bubble Sort, Selection Sort

Time complexity: O(n^2) - Quadratic Time Complexity

1. Polynomial-time algorithms for specific problems

Time complexity: O(n^k) - Polynomial Time Complexity

1. Recursive Fibonacci (naive implementation)

Time complexity: O(2^n) - Exponential Time Complexity

**Number 5**

Abstract Data Type (ADT) is a high-level description of set of operation that can be performed on a data structure, without specifying the details of how these operations are implemented.

Example in Java:

// Example of an ADT: Stack

public interface Stack<T> {

void push(T item);

T pop();

T peek();

boolean isEmpty();

}

// Implementation of Stack using an array

public class ArrayStack<T> implements Stack<T> {

private T[] array;

private int top;

public ArrayStack(int capacity) {

array = (T[]) new Object[capacity];

top = -1;

}

public void push(T item) {

if (top == array.length - 1) {

// Handle stack overflow

} else {

array[++top] = item;

}

}

public T pop() {

if (isEmpty()) {

// Handle stack underflow

return null;

} else {

return array[top--];

}

}

public T peek() {

if (isEmpty()) {

// Handle empty stack

return null;

} else {

return array[top];

}

}

public boolean isEmpty() {

return top == -1;

    }

}

**Number 6**

|  |  |  |
| --- | --- | --- |
| **Feature** | **List** | **Array List** |
| Interface vs Class | List is an interface in the java.util package. | ArrayList is a class in the java.util package, implementing the List interface. |
| Dynamic Sizing | Can be implemented by different classes such as ArrayList, LinkedList, etc. | Specifically implemented as a dynamic array. |
| Memory Overhead | More memory overhead as it could be implemented by a linked list, array, etc. | More memory-efficient due to a dynamic array implementation. |
| Performance | Can have varying performance depending on the underlying implementation. | Generally offers better performance for random access and traversal compared to other List implementations like LinkedList. |
| Insertion and Deletion | Slower for inserting or deleting elements in the middle of the list. | Faster for inserting or deleting elements in the middle due to array resizing overhead only for appending. |
| Initial Capacity | No initial capacity is specified in the List interface. | ArrayList allows you to specify an initial capacity for better performance if the size is known in advance. |
| Type Safety | No direct type safety; elements are added using the add method. | Supports type safety, and the generic type is specified during declaration. |
| Null Elements | Allows null elements. | Allows null elements. |
| Synchronization | Not synchronized. Use Collections.synchronizedList for synchronization. | Not synchronized. Use Collections.synchronizedList for synchronization. |
| Iterating through Elements | Uses iterators or enhanced for loop. | Uses iterators or enhanced for loop. |

**Number 7**

import java.util.ArrayList;

public class Main {

public static void main(String[] args) {

ArrayList<Integer> num = new ArrayList<>();

// Adding elements

num.add(12);

num.add(25);

num.add(34);

num.add(46);

// Print list

System.out.println(num);

// Remove 25

num.remove(1);

// Print list after change

System.out.println(num);

}

}