a. The algorithm is O(n2) because the outer loop runs n times and for each iteration of the outer loop, the inner loop runs n times. So, the total number of executions are n \* n = n2.

b. The algorithm is O(n) because the outer loop runs n times and the inner loop always runs 2 times for each iteration of the outer loop. So, the total number of executions are 2 \* n = 2n.

c. The total number of executions is n\*(n+1)/2, which means the algorithm is O(n2).

d. The total number of executions is n\*(n+1)/2, which means the algorithm is O(n2).

1. The contents of anArray after each loop is that each number after the index i in the array is changed to the number in index 3 because i = 3.

Output:

[0, 1, 2, 3, 3, 5, 6, 7]

[0, 1, 2, 3, 3, 3, 6, 7]

[0, 1, 2, 3, 3, 3, 3, 7]

[0, 1, 2, 3, 3, 3, 3, 3]

The contents of anArray after each loop is that the last index is changed to the number of the index before it and it continues by moving to the left so the next loop the seconds last index is changed to the number before it. This last loop will change the number for index 4 because the value of i has to be more than 3.

Output:  
[0, 1, 2, 3, 4, 5, 6, 6]

[0, 1, 2, 3, 4, 5, 5, 6]

[0, 1, 2, 3, 4, 4, 5, 6]

[0, 1, 2, 3, 3, 4, 5, 6]

a. Time Complexity = O(n)

The loop iterates over each element of the array only once so the time complexity is O(n).

Space Complexity = O(1)

The algorithm uses constant amount of extra space so the space complexity is O(1)

b. Time Complexity = O(r1⋅c2⋅c1)

The time complexity is calculated by the number of operations in terms of the matrix

dimensions, in which the outer loops run r1 times, the middle loops run c2 times, and the innermost loop runs c1 times.

Space Complexity = O(r1⋅c2)

The product matrix has dimensions r1 x c2, so the time complexity is O(r1⋅c2).

c. Time Complexity = O(n)

The loop iterates over the variable n, which means that it does a constant number of operations in each iteration, which means that the time complexity is O(n) for each loop.

Space Complexity = O(1)

The space complexity is O(1) because the variable result is updated within each loop, but is overwritten in each iteration so the space complexity is constant.

d. Space Complexity = O(log n)

The space complexity continues as long as i/2 > 0, so the space complexity is O(log n).

Time complexity = O(1)

Time complexity is O(1) because there are no variables, in which the size depends on the input.

1. **Constant Time Complexity (O(1)):**

Retrieving an element from an array with a known index.

**Logarithmic Time Complexity (O(log n):**

- Binary Search in a sorted tree or array

**Linear Time Complexity (O(n)):**

- Linear Search through unsorted array

**Linearithmic Time Complexity (O(n log n)):**

- Merge Sort

**Quadratic Time Complexity (O(n^2)):**

- Bubble Sort

**Cubic Time Complexity (O(n^3)):**

- Matrix Multiplication

**Exponential Time Complexity (O(2^n)):**

- Recursive algorithm for computing Fibonacci numbers.

**Factorial (O(n!)):**

- Brute-force solution for the Traveling Salesman Problem.

1. An abstract data type is a high-level description of a set of data and the operations that can be performed on that data. It defines a logical model for data and operations but does not specify the implementation details. The implementation is left to the concrete data structures or classes that instantiate the ADT.

Example of Queue:

import java.util.\*;

import java.util.Queue;

import java.util.LinkedList;

public class Stack {

public static void main(String[] args) {

java.util.Stack<Integer> stk = new java.util.Stack<>();

stk.push(78);

stk.push(420);

stk.push(113);

Boolean result = stk.empty();

System.*out*.println(stk);

System.*out*.println(stk.peek());

System.*out*.println(result);

stk.pop();

System.*out*.println(stk);

stk.search(78);

System.*out*.println(stk);

Queue<String> qs = new LinkedList<>(); {

qs.add("Banana");

qs.add("Apple");

System.*out*.println(qs);

System.*out*.print("Queue after removal: " + qs);

System.*out*.println("The first element of the queue: " + qs.peek());

qs.add("banana");

qs.remove();

qs.add("Mango");

System.*out*.println("The first element of the queue after removal: " + qs.remove());

}

}

}



| Feature | List | Interface |
| --- | --- | --- |
| Implementation | Interface | Class |
| Resizeability | Dynamic resizing | Dynamic resizing |
| Access Speed | Slower because of abstract methods | Faster due to direct access to array elements |
| Memory Overhead | Less | More due to internal array and additional features |
| Usage | General purpose usage | Typically used when fast element access is required |

1. 