Time Complexity Calculations

i)
```
private String append(char c,int n)
{
        String sequence = "";
        for(int i=0;i<n;i++)
        {
            sequence += c;
        }
        return sequence;
}
```

This program has an append method that returns a final sequence variable value when the user inserts n value iteratively into the program. In each iteration, that variable is updated, and a new string is added.

The first initialization statement have an empty string with the value " ", it takes the constant time complexity **O(1).**

This loop is iterative n times when the user is inserted and the "i" value is increment by one in each iteration. (Start 0 and n times iterative) It takes the linear time complexity **O(n).** Inside the statement is not a constant factor, because the inner loop is iterative n times each time the sequence variable is updated and a new string is added to old string. That operation takes linear time complexity **O(n)**. The line return sequence returns the final sequence string. This operation takes constant time, **O(1).**

Overall time complexity = ~~O(1)~~ +{ O(n) x O(n) } + ~~O(1)~~

= $\mathbf{O(n^2)}$

(Ignore the constant factors

ii)
```
boolean ContainsValue(int array[], int value)
{
    for(int i=0;i<array.length;i++){
        if (array[i] == value)
        {
            return true;
        }
    }
    return false;
}
```

This program is a linear search algorithm for finding a specific value in a given array. If that value matches with the corresponding array value return true, else false.

The loop is iterative given array length times and, in each iteration, the '**i**' value is incremented by one. (Start 0 until it goes the length of array values times) That operation takes linear time complexity as **O(n).** The inside if the condition is checked user inserted value matches to array[i] element and return true. that operation takes constant time for each value **O(1).** Finally, it will return true when both values are matched it is a constant time complexity **O(1)**. The last return false; statement is the return constant value that has constant time **O(1).**

Overall time complexity = O(n) + ~~O(1)~~ + ~~O(1)~~ + ~~O(1)~~

= **O(n)**

(Ignore the constant factors)

iii)
```
void jump(int array[])
{
   int n = array.length;
   for (int i = 0; i < n; i++)
   {
      for (int j = 1; j < n; j = j*2)
      {
          System.out.println(array[i] * array[j]);
      }
   }
}
```

This program purpose of returning the combination of array values with inner and outer loop corresponding indexes (i and j)

The first initialization statement is assigned to an array length that is fixed value and it takes constant time complexity **O(1).** The outer loop iterative n times (n means the length of a given array) starts with 0 element, in each iteration the **'i'** value is increment by one and that takes linear time complexity **O(n).**

The outer loop starts one: not zero, then it is iterative (n-1) times, and each iteration the corresponding **'j'** value is multiplied by 2. That operation takes logarithmic time complexity, for example when the user inserts 8 values to a given array then the inner loop is iterative **$\log_2(8)$ times → 3 times.** So that loop takes **O(logn).** Inside the print statement returns the multiplication of array[i] and array[j] elements and that have constant value **O(1).**

Overall time complexity = ~~O(1)~~ + { O (n) x O(logn) } + ~~O(1)~~

$$= \mathbf{O(n*logn)}$$

(Ignore the constant factors)

i)

```
int product(int arr[])
{
    int result = 1;
    for (int i = 0; i < arr.length; i++)
    {
        result = result * arr[i];
    }
    return result;
}
```

The purpose of this program is output the value of the user given array elements are multiply by the next element until it reaches the end element for an array.

This program the first initialization statement has a variable and its assigns to one. That operation takes constant time complexity because it has a fixed value of **O(1).** The for loop iterative array length number of times with a starting value of zero and each iteration the 'i' value is increment by one.That operation takes linear time complexity **O(n).**As well as the inside statement is added each iteration wise corresponding array element of given array. That variable have constant value always then it takes constant time complexity like **O(1).** Finally the last statement has a return result ; and it takes constant value to output **O(1).**

Overall time complexity = ~~O(1)~~ + O (n) + ~~O(1)~~ } + ~~O(1)~~

= **O(n)**

(Ignore the constant factors)

The user given array size is proportional to number of times that loop iterates thought the program.

ii)

```
double standardDeviation (int array[])
{

      double sum = 0,avg;
      for(int i=0;i<array.length;i++)
      {
         sum = sum + array[i];
      }
      avg = sum/ array.length;
      sum = 0;
      for(int i = 0;i < array.length;i++)
      {
           sum += Math.pow((array[i]- avg),2);
      }
      avg =sum/(array.length - 1);
      double deviation = Math.sqrt(avg);
      return deviation;
}
```

This program's purpose is to print the standard deviation of user-given array elements.

Initialization: The variables sum and average are initialized, which takes constant time, **O(1).**

First loop: The first loop iterates over the array to calculate the sum of all elements. This loop runs n times, and each iteration performs a constant time operation (adding an element to the sum). Therefore, the time complexity of this loop is **O(n).**

Calculation of average: The average is calculated by dividing the sum by the array length. This is a constant time operation **O(1).**

Second loop: The second loop iterates over the array again to calculate the sum of the squared differences between each element and the average. Similar to the first loop, this loop also runs n times and performs a constant time operation. Therefore, the time complexity of this loop is **O(n).**

Calculation of standard deviation: The standard deviation is calculated by taking the square root of the average of the squared differences. This is a constant time operation.

Overall time complexity = ~~O (1)~~ + O (n) + O(n) + ~~O(1)~~

= **O(n)**

(Ignore the constant factors)

Since the dominant operations in the function are the two loops, which both have O(n) time complexity, the overall time complexity of the standard deviation function is **O(n)**.

iii)

```
int find(int array [], int key)
{
    int lower = 0, upper = array.length - 1;
    while (lower <= upper) {
        int mid = lower + (upper - lower) / 2;

        if (array[mid] == key)
            return mid;
        if (array[mid] < key)
            lower = mid + 1;
        else
            upper = mid - 1;
    }
    return -1;
}
```

The above program is a binary search algorithm for finding specific items in the given array.

The first initialization statements have created two variables like, lower and upper variables, and assigned their 0 and the length of the given array respectively. So it takes constant time complexity then **O(1).**

The while loop is iterative when lower part is greater than the upper part. In each iteration, the inside statements are performed as follows,

The middle value calculates statement takes a constant value because it has an integer number then it takes constant time complexity **O(1)** But each iteration the original array is divided into 2 parts and consider one part that operation is takes logarithmic fashion **O(logn).** The if the condition is checked array's middle value is equal to the search value it takes constant time **O(1).**If not the checks given search value is greater than or less than the middle value of the array and if it is true return their lower part or upper parts. So both take constant time **O(1).**

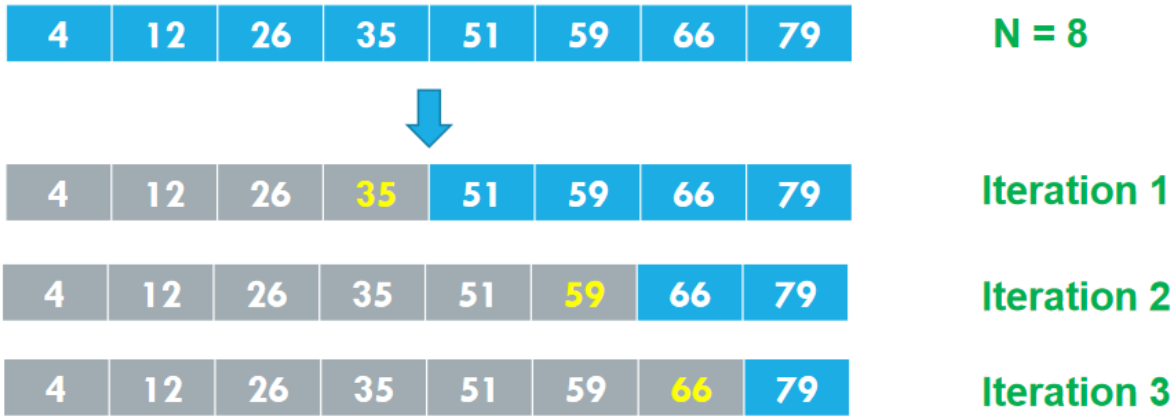
The function returns either the index of the key or -1, taking constant time, **O(1).**

The loop's iterations are halved in each step, effectively dividing the search space in half. This behavior is characteristic of a logarithmic function.

Overall time complexity = ~~O(1)~~ + ~~O(1)~~ + O(logn) + ~~O(1)~~+~~O(1)~~

= **O(logn)**

(Ignore the constant factors)



Total time = (3 * time for one iteration) + other steps

= (3 * C1) + C2

C1,C2 are constants !

Total time → 3 → $\log_2 8$ ⟹ $\log_2 N$

i)

```
public void orderedPairs(int arr[])
{
  System.out.print("Items in the array:");
  for(int i = 0; i < arr.length; i++)
  {
    System.out.print(arr[i] + "\t");
  }

  System.out.println("All pairs in the array:");
  for(int i = 0; i < arr.length; i++)
  {
    for(int j = 0; j < arr.length; j++)
    {
      System.out.println(arr[i] + "," + arr[j]);
    }
}
```

This program's purpose of output the ordered pairs of given array elements using nested loops.

The first statement is used to print the items in the array it takes constant time **O(1).** The first loop is iterative each of elements in the array start 0 to the end element as well as every iteration current array element is printed. However, the loop takes linear time complexity **O(n)** and the inside statement takes constant time **O(1).** The other statement is used to print the items in the array in ordered pairs, so it takes constant time **O(1).**

Finally, two loops are iterative for each of the array elements with a starting index of 0 to the end element. So both loops take linear time complexities **O(n) and O(n)**. The inside print statement is output into a set of ordered pairs of given array elements and it takes constant time **O(1).**

Overall time complexity = $\cancel{O(1)} + \cancel{O(n)} + \{ O(n) \times O(n) \} + \cancel{O(1)} + \cancel{O(1)} + \cancel{O(1)}$

$= \mathbf{O(n^2)}$

(Ignore the constant factors)

iii)

```
public int count(int n)
{
    int operations = 0;
    int i = 1;
    while(i < n)
    {
        i = i * 2;
        operations++;
    }
    return operations;
}
```

This program represents calculating a number of operations that can perform this. It will be affected by two things such as ‘n’ and ‘i’ values. The program will iterate n number of times and in each iteration, the I value is multiplied by 2. So these first two initialization statements have two integer variables assigned values. It takes constants time complexity **O(1).**The while loop iterates n number of times and it takes linear time complexity **O(n).**

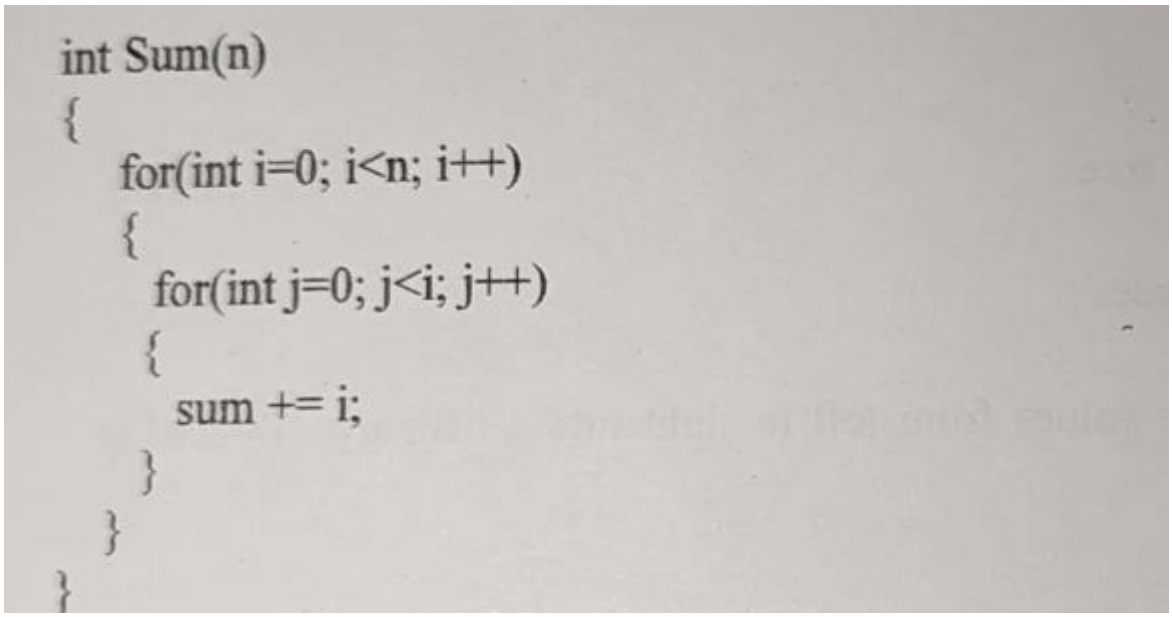
The inside i=i*2 statement is each iteration I value is multiplied by 2. It is like logarithmic time complexity **O(logn).** Finally the return statement outputs a number of operations that have taken constant time **O(1).**

Overall time complexity = ~~O(1)~~ + ~~O(1)~~ + O(logn) + ~~O(1)~~+~~O(1)~~

= **O(logn)**

(Ignore the constant factors)

(If n=8 → $\log_2(8) = 3$ → 3 times loop executes)



```
int Sum(n)
{
    for(int i=0; i<n; i++)
    {
      for(int j=0; j<i; j++)
      {
        sum += i;
      }
    }
}
```

The given code snippet calculates the sum of the first n natural numbers using nested loops. Here's a breakdown of its time complexity:

Outer Loop:

- Iterates from i = 0 to i < n, executing n times.

Inner Loop:

- Iterates from j = 0 to j < i, executing i times for each value of i.

Total Number of Iterations:

- The total number of iterations is the sum of i for i = 0 to n-1, which can be calculated using the formula:
- sum = (n-1) * (n-1 + 1) / 2 = n * (n-1) / 2

Overall time complexity = {O (n) x O (n) }+ + ~~O~~(1)

= $\mathbf{O(n^2)}$

(Ignore the constant factors)

```java
public class Fibonacci {
    public static int fibonacci(int n) {
        if (n <= 1) {
            return n;
        } else {
            return fibonacci(n - 1) + fibonacci(n - 2);
        }
    }
```

The time complexity of the given recursive Fibonacci function is **$O(2^n)$** because the function makes two recursive calls for each input: fibonacci(n-1) and fibonacci(n-2). This means that for each call, the function branches out into two more calls, forming a binary tree of recursive calls. As a result, the total number of function calls grows exponentially, leading to a very inefficient algorithm for larger values of n. For example, calculating fibonacci(5) would require computing fibonacci(4) and fibonacci(3), but both of these also recompute many smaller Fibonacci numbers, causing redundant calculations.

Overall time complexity = **$O(2^n)$**