Algorithm Analysis and Program Complexity

Empirical analysis of an algorithm: One way to determine an algorithm's approximate time complexity is to program it, run the program, and measure how long it takes to run. However, this can give misleading results since running the same program on difference machines and platforms may give varying results. It would be hard to establish a benchmark for the resources that a program consumes in terms of time.

Algorithm Analysis:

- A more neutral way to measure a program's performance by examining its code or pseudocode and roughly count the number of statements that are executed.
- Apply techniques to mathematically approximate the performance of various computing algorithms.
- Algorithm analysis is an important tool in computer science.
- One of the fundamental principles of science in general is that we can make predictions and hypotheses using formal models, which we can then test by experimentation.

Counting Instructions:
When it comes to algorithm complexity analysis, we want to look not only at the number of instructions being performed, such as variable assignments and arithmetic operations, but actually how the number of instructions increases as the algorithm's input data size increases does.

When our algorithm is dealing with a fixed number of operations or has a small input, like an array of size three elements, how many instructions there are doesn't matter too much. For example, see these algorithms:
Note: Since we are analyzing algorithms, we will use pseudocode in these examples:

```
Example 1:
def sum(int a, int b):

     int sum = 0 // assignment
     sum = a + b;
     return sum

Example 2:
def sum(int array[], int length):// length == 3
```

```
int sum = 0 // assignment
int index = 0 // assignment

while(index < length): // comparison (1)
     sum = sum + array[index] //access, add, assign (2)
     index = index + 1 // add and assign (3)

return sum (4)
```

In order to arrive at a categorization for time complexity, let us take the approach of:

- counting the number of operations
- determining a formula that represents the count
- approximating this formula by focusing on the dominant terms
- categorizing the resulting formula.

We would make simplifying assumptions that each operation like assignment, arithmetic and logical evaluations, function calls etc are assigned "one unit of time" for each. Using this process for Example 1 we arrive at 3 units. Example 2 has a loop. The while loop, (comparison (3)) is executed 4 times, the statements(2) and (3) are each executed 4 times and the return statement gets executed 1 time. That's a total of 13 units of time.

A similar calculation can be done for a method without an input array but which has repetition structures like while and for loops.
Usually loops are used to iterate over elements in containers so let us stay with the array parameter scenario.

Generating a formula

Let us ask if unlike example 1 and example 2, what if the program or function is processing an array of data elements of unknown size? Array traversals require loops and the number of times this kind of loop executes increases with the length of the array. The while loop in example 2 has about five or six instructions, per iteration.

So each iteration requires, say 6 instructions. How many iterations are there are? Equal to the array length!

If the array length is 4, then the algorithm has around: 2 + (6 instructions per loop) * (4 iterations) = 2 + 24 = 26 instructions.

If we represent the number of iterations/size of Array as 'N', then the number of instructions is:

2 + (6) * (N) = 2+6N

Since each operation is considered to take one unit of time, this is also the total units of time
Now, this is like a linear equation, isn't it? Just like y = 2+6x or y = 2+6N.

The term 'linear' here is important because it tells us how big 'y' is as we increase 'N'. In this equation, it's a linear increase, meaning that 'y' has a proportional, steady growth as 'N' increases.

If we graphed this it would look like a straight line with the x axis representing the size of the array and the y axis representing the number of instructions or the units of time.

Look at this table:

N	1	10	100	1000
y	8	62	602	60002

Simplifying or Approximating the formula
We could use some further simplifications and approximations to this equation as the size of the database gets larger.

Notice that the constant value '2' in y = 2 + 6N is not significant as N gets larger, so we can instead approximate the linear equation via a symbol called 'O' as y = O(6N).

This symbol is called Big-O notation, and helps us describe the growth of 'y' in an approximate way, without extraneous details.

Categorizing into Big O bins

Our linear equation is in a category of Big-O called 'linear' time algorithms, because it is a linear equation. In fact, all linear time algorithms can be represented by y = O(N). O(N) is the category of linear algorithms.

There are other categories of complexity also! Some algorithms can be represented by quadratic equations like $y = x^2$, or $y = O(N^2)$. Others are called logarithmic, like y = O(log N).

Here are some basic categories, in order of their time complexity:

O(1): constant time complexity. These algorithms don't require more time even if the input increases, i.e. the time required is a constant, not a variable. They are the fastest algorithms.

O(log n): logarithmic time complexity. These algorithms are not as fast, but still very quick.

O(n): linear time complexity.

O(n^2): quadratic time complexity.

O(2^n): exponential time complexity. These are pretty slow!

Here is an example of a constant time algorithm. The time required doesn't change even if we increase the input.

```
def increment(int value):
     return value + 1 // add
```

No matter what the size of the input value is, the algorithm always does one or two instructions. It will never do 100 or 1000 instructions, even if the input value is 100 or 1000 itself.

Q1) Give an example of a constant time algorithm, i.e. one in which the number of instructions is not affected by the size of the input value(s).

Q2) Here is another example of a constant time algorithm. Label each instruction and provide the total number of instructions. You don't need to count return statements or include the method header.

```
def isNegative(int value):
     if value < 0:
          return True
     else:
          return False
```

Now we will look at some linear time algorithms, identify their complexity, and categorize them as y= O(N) algorithms.

Q3) For the following linear time algorithm, label each instruction in it, then count the total number of instructions, formulate the result as a linear equation, i.e. y = 3 + 7n. Do not count return statements, but do count array access.

```
def maximum(int array[], int length):

     int max = array[0]

     int i = 0
     while(i < length):

          if array[i] > max:
               max = array[i]

          i = i + 1

     return max
```

Q4) Do the same for the following algorithm.

```
def linearSearch(int array[], int length, int target):

     int i = 0
     while(i < length):
          if array[i] == target:
               return True

          i = i + 1
```

```
     return False
```

Now that we know how to estimate an algorithm's complexity, captured by the O() expression, let's try to reduce algorithms from a higher complexity class to a lower one.

For example, a simple operation commonly done in Java is accessing an element stored in an array, which is a constant time operation, O(1). This is because the operation is not affected by how large the array is. For example:

```
//create an array and fill it
int[] array = new int[3];

array[0] = 2;
array[1] = 4;
array[2] = 6;

// access element as an O(1) operation
System.out.println("Second element: " + array[1]);
```

We could also use a linear-time approach, O(n), to get the desired element instead. We would use a loop and counter as such:

```
// use a loop to access the element, as O(n) operation

for(int i=0; i < array.length; i++){

     if(i == 1){
          System.out.println("Second element:" + array[i]);
     }
}
```

Q5) Provide both an O(1) solution and O(n) solution to access the final element in a given Java array. You may assume the array is not empty.

```
public static void constantTimeAccess(int[] arr, int length){
     //
}

public static void LinearTimeAccess(int[] arr, int length){
     //
```

```
}
```

Q6) Here is a linear time O(n) solution for returning the sum of the first and last element of a non-empty array. Provide the constant time O(1) solution.

```
public static int LinearTimeAdd(int[] arr){

     // O(n): try to loop over array elements
     int sum=0;
     for (int i = 0; i < arr.length; i++) {
        if (i==0) sum+=arr[i];
        else if (i==arr[arr.length-1]) sum+=arr[i];
     }

     return sum;
}

public static boolean ConstantTimeAdd(int[] arr) {
     //
}
```

Q7) Provide both O(1) and O(n) solutions for updating the array element at the specified index with the provided value.

```
public static void constantTimeArrayUpdate(int[] arr, int index,
int value){
     //
}

public static void LinearTimeArrayUpdate(int[] arr, int index,
int value){
     //
}
```

Summary of Rules for counting instructions, determining and approximating corresponding equations.

Using simple rules, we can extrapolate the runtimes of larger and more complex pieces of code.

Rule 1: The runtime of a group of statements in sequential order is the sum of the individual runtimes of the statements:
Rule 2: The runtime of a loop is roughly equal to the runtime of its body times the number of iterations of the loop. For example, a loop with a body that contains *K* simple statements and that repeats *N* times will have a runtime of roughly (*K* * *N*):
Rule 3: The runtime of multiple loops placed sequentially (not nested) with other statements is the sum of the loops' runtimes and the other statements' runtimes: N+M+3
Rule 4: The runtime of a loop containing a nested loop is roughly equal to the runtime of the inner loop multiplied by the number of repetitions of the outer loop: eg. 3*M*N
Rule 5: When you have a N^3 + N^2 situation, consider the large power to dominate

Sample Problems and Solutions that use that above rules:

The following methods each take an array with n elements as input. For each, put a box around any statement that contributes to the lead term of the work formula. Exactly how many times does each boxed statement get executed? What is the big-oh running time of each method?

1)

```
public static int foo1(int[] arr) {
    int sum = 0;
    for (int i=0; i<arr.length; i++) {
        sum += arr[i];
    }
    return sum;
}
```

2)

```
public static int foo2(int[] arr) {
    int sum = 0;
```

```
    for (int i=0; i<arr.length; i++) {
        for (int j=0; j<arr.length; j++) {
            sum += arr[i];
        }
    }
    return sum;
}
```

3)

```
public static int foo4(int[] arr) {
    int sum = 0;
    for (int i=0; i<arr.length; i++) {
        for (int j=0; j<10; j++) {
            sum += arr[i];
        }
    }
    return sum;
}
```

4)

```
public static int foo3(int[] arr) {
    int sum = 0;
    for (int i=0; i<arr.length; i++) {
        for (int j=i; j<arr.length; j++) {
            sum += arr[i];
        }
    }
    return sum;
}
```

5)

```
// pre: arr.length is a power of 2
public static int foo5(int[] arr) {
    int sum = 0;
    for (int i=1; i<=arr.length; i*=2) {
        sum += arr[i];
    }
    return sum;
}
```

Answers

In each of these, the focusis on the most executed statement, but the way this worksheet is written, other statements contributing to the dominant term would be acceptable too.

1) `i<arr.length` gets executed the most, n+1 times (true n times, false 1 time). O(N).

2) The inner for-loop gets started n times, and each time the inner for-loop runs, its test j<arr.length gets executed n+1 times.
So, in total, j<arr.length executes $n^2 + n$ times. So, O(N^2).

3) The inner for-loop gets started n times, and each time the inner for-loop runs, its test j<10 gets executed 11 times.
In total, j<10 executes 11n times. So, O(N).

4) The inner for-loop gets started n times, but each time it gets started it loops one fewer time than the time before. The first time, the inner for-loop loops n times (and so the j<arr.length test occurs n+1 times). The second time the inner for-loop starts, it loops n-1 times (and so the j<arr.length test occurs n times). The last time the inner for-loop gets started, i=arr.length-1, and so the inner-loop loops just once (and so the j<arr.length test occurs 2 times). Summing the number of times the test j<arr.length occurs we get: (n+1) + (n) + (n-1) + ... + 2 = $n(n+3)/2 = 0.5n^2 + 1.5n$ which is O(N^2).

5) Starting with 1, how many doublings does it take to get to n? Since we are told n is a power of 2, the answer is the base-2 logarithm of n. So, the test i<=arr.length occurs exactly log(n) + 1 times, which is O(log NS).