DSA Tutorial Week 2 - Algorithm Analysis

Find All Source Code on the DSA Tutorial Github: [DSA-Tutorial-Exercises]()

Problem 1 - Algorithm Analysis

Give the tightest possible upper bound for the worst-case runtime for each of the following functions in Big-Oh notation in terms of the variable n. Choose your answer from the following (not given in any order), each of which could be re-used (could be the answer for more than one of a. – d.):

$O(1)$, $O(n^2)$, $O(n \log n)$, $O(n)$, $O(2^n)$, $O(n^3)$, $O(\log n)$, $O(n^4)$, $O(n^5)$, $O(mn)$

The correct answer to each can be revealed by selecting the black text below the pointers. (If you printed the tutorial this won't work of course, in that case, godspeed!).

a.

```
void silly(int n, int x, int y) {
    if (x < y) {
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n * i; ++j)
                System.out.println("y = " + y);
   } else {
        System.out.println("x = " + x);
   }
}
```

Pointers:

1. **Nested Loops:** Observe how the loops are nested and how their limits are defined. The innermost loop runs ***n*i*** times, which changes with each iteration of the outer loop.
2. **Growth Rate:** Consider the effect of the outer loop running ***n*** times and the inner loop's limit increasing linearly with each iteration of the outer loop. The multiplication of these two factors will give you the overall growth rate.

Answer [redacted]

b.

```
int silly(int n, int m) {
    if (n < 1 || m < 1) return n;
    else if (n < 100)
        return silly(n - m, m);
    else
        return silly(n - 1, m);
}
```

Pointers:

1. **Recursive Calls:** Focus on how the function calls itself recursively and under what conditions. The depth of the recursion is key. Perhaps you can draw a recursion path/tree to find out this depth.
2. **Worst-Case Scenario:** Think about the ***worst-case input*** that would cause the ***maximum number of recursive calls***. This will help identify the upper bound of the runtime.

Answer [redacted]

c.
```
void foo(int number) {
      int steps = 0;
      while (number > 1) {
            number = number / 2;
            steps++;
        }
       System.out.println("Total steps taken to reach 1: " + steps);
}
```

Pointers:
1. **Halving the Problem:** Recognize how the number is being halved in each iteration. This should hint at the logarithmic nature of the problem.
2. **Number of Iterations:** Consider how many times you can divide ***n*** by ***2*** before it becomes ***1*** or less. This directly relates to the logarithmic complexity.

Answer [redacted]

d.
```
public static int fib(int n) {
      if (n <= 1) {
            return n;
      }
      return fib(n - 1) + fib(n - 2);
}
```

Pointers:
1. **Recursive Pattern:** Look at the structure of the recursive calls. Notice how each call spawns two more calls. Perhaps to understand the problem better, you can draw a recursion tree to analyze how the number of calls increases with each recursive call. Also use this representation to find the total depth of the tree (in terms of ***n***).
2. **Growth:** Think about how the number of calls grows with increasing n. Each step essentially doubles the number of calls, leading to exponential growth.

Answer [redacted]

e.
```
public static int middle(int[] array) {
    if (array == null || array.length == 0) {
        throw new IllegalArgumentException("Array cannot be null or empty");
    }
    return array[array.length / 2];
}
```

Answer: [redacted]

f.

```
public static int[][] pairs(int[] array1, int m, int[] array2, int n) {
    int[][] productPairs = new int[m][n];
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            productPairs[i][j] = array1[i] * array2[j];
        }
    }
    return productPairs;
}
```

Answer: [redacted]

Reflections

- Reflect on your understanding of Big-Oh notation. How does it help in comparing different algorithms?
- Think about how your approach to analyzing loops has evolved. Are you more comfortable with determining their impact on the overall runtime?
- Reflect on your understanding of recursive functions. How do the number and nature of recursive calls affect the runtime?
- Consider how you managed to simplify and understand the complexity of nested structures. What strategies worked best for you?
- Think about how these skills translate to real-world coding. How might these insights affect your approach to writing and optimizing code?

Problem 2 - Find the Duplicates

a. Describe an algorithm that checks if an array of integers contains at least ***k*** duplicates of any element. The method should have a time complexity of **O(n^2)** and a storage complexity of **O(1)**. Discuss how the algorithm should work and write it in **pseudocode**.

   Pointers:
   1. Considering the storage complexity, can you use, for instance: *sets*, *lists*, or any other data structure?
   2. Given the ***O(n^2)*** complexity, what do you expect a solution to look like?
   3. **Input:** The method should take an array of integers and an integer ***k*** as input.
   4. **Output:** Return `true` if any element appears at least ***k*** times, otherwise return false.

   ### Reflections
   1. Consider how the loops, each running up to ***n*** times, contribute to the ***O(n^2)*** complexity.
   2. Explore different scenarios, such as when ***k*** is larger than the array size or when ***k*** equals 1.
   3. Analyze the algorithm's behavior with unique-element arrays versus arrays with multiple duplicates.

   A Java example is available in **Duplicates.java**. Though try to write it yourself first.

b. *Improve* the algorithm (written in a.) to check if an array contains at least ***k*** duplicates of any element, but this time with a time complexity of ***O(n)***. This time, you may use ***O(n)*** storage complexity. You will require an extra data structure to solve this problem, when writing the **pseudocode**, you may use this data structure as a given (i.e. you do not have to implement the data structure).

   Pointers:
   1. Understand how an extra data structure can reduce the need for nested loops.
   2. What requirements do we have for this extra data structure? What do we store in it? How about its complexity?
   3. Think of how iterating through the array just once (single loop) contributes to achieving the decrease in complexity.

   A Java example is available in **Duplicates.java**. Though try to write it yourself first.

   ### Reflections
   1. Describe why your algorithm achieves ***O(n)*** time complexity.
   2. Think about how your algorithm handles cases such as **k** greater than the array size, **k** equals **1**, and when the array contains all unique elements.

Problem 3 - Detecting Duplicate Strings

Implement a method that checks for duplicates in an array of strings. The core of this assignment is to create a ***custom hash function*** for strings and use it in a hash set implementation to efficiently track duplicates.

Requirements:

1. **Custom Hash Function:** Design and implement a hash function that converts strings into hash values. Explain your choice of hashing strategy.
2. **Hash Set Creation:** Use your hash function within a hash set structure to manage the strings.
3. **Duplicate Detection:** Traverse the array, using the hash set to detect duplicates.
4. **Input:** An array of strings.
5. **Output:** Return true if any duplicates are found; otherwise, return false.

Start from this code:

```
function customHashFunction(string, size):
    // TODO Implement a simple but effective hash function for strings

function hasDuplicateStrings(array):
    create hashSet using customHashFunction
    for each string in array:
        if hashSet contains hash of string:
            return true
        else:
            add hash of string to hashSet
    return false
```

Start by discussing the task, what is required? What is the problem? How could you approach the task? Write your solution(s) in **pseudocode.**

Pointers:

1. What characteristics make a hash function effective for strings?
2. How does the hash function impact the distribution of strings in the hash set?
3. Consider the role of collision resolution in your hash set and how it affects performance.
4. A naive approach as a starting point, you might consider a simple hash function like summing the **ASCII** values of the characters in the string. Remember, a good hash function for strings should distribute hashes uniformly across a wide range.
5. Research **'string hashing'**, **'polynomial rolling hash function'**, and **'hash code for strings'**. These terms will help you understand common methods for converting strings to hash values.

Reflections

- **Understanding Hash Functions:** Reflect on what makes a good hash function for strings. A good hash function should:
    1. Distribute hash values uniformly to minimize collisions.
    2. Be efficient to compute.
    3. Generate different hash values for different strings as much as possible (though collisions are inevitable in practice).

- Understand that **collisions** occur when different strings produce the same hash value. What is your responsibility in dealing with this issue?
- Do you think that different applications require different types of hashing? How about collision resolution?
- If you were to optimize your algorithm, what aspects would you focus on? Would you modify the hash function, change the way collisions are handled, or adjust the hash table size?

Problem 4 - Collision Resolution (Optional Challenge)

The file **HashMap.java** contains a partial implementation of a Dictionary Abstract Data Type (ADT) using a hashmap. However, the implementation lacks a crucial component: collision resolution.

a. Implement **Chaining** for collision resolution:
   i. Make a copy of the original `HashMap.java`.
   ii. Modify and complete the implementation by introducing the chaining method to resolve collisions. Feel free to adapt any part of the existing code to accommodate chaining.
   iii. Specifically, implement the put and get methods to manage collisions using chaining. Ensure each index of the array stores a linked list (or a similar structure) of entries to handle collisions.

b. Implement **Double Hashing** for collision resolution:
   i. Make a copy of the original `HashMap.java`.
   ii. In this new version, replace the chaining method with double hashing for collision resolution. Double hashing involves using a second hash function to determine a new index when a collision happens, aiming to reduce clustering that might occur with chaining.
   iii. Alter the put and get methods to accommodate double hashing. You will also need to implement an additional hash function that differs from the primary one.

Reflection

- **Performance:** How do the two methods behave under heavy load or numerous collisions? Which method handles clustering more effectively?
- **Memory Usage:** Chaining typically requires more memory due to linked list pointers. How does this compare to the potentially increased array size for double hashing?
- **Complexity**: Consider the complexity of implementing and understanding each method. Which method might be more suitable for different types of applications?
- **Use Cases:** In what scenarios might one method be preferred over the other? For instance, chaining might be better for small datasets with few collisions, while double hashing might excel in larger datasets.

Refer to the provided **DoubleHashingHashMap.java** and **ChainingHashMap.java** for sample implementations and additional guidance.