Homework 4 - Analysis

In this homework, we are going to work to become comfortable with the mathematical notation used in algorithmic analysis.

Problem 1: Quantifiers

For each of the following, write an equivalent *English statement.* Then decide whether those statements are true if x and y are integers (e.g., they can be any integer). Then write a convincing argument to prove your claim.

1. $\forall x \, \exists y : x + y = 0$

Equivalent English Statement: "For every value of x, there exists a value of y such that their sum equals zero."

Claim: The statement is true for all integers x and y.

Argument:

Let's consider any integer value for x. We need to find a corresponding integer value for y such that x + y = 0.

If x is positive, we can choose y to be its negative counterpart. For example, if x = 3, then y = -3. The sum x + y = 3 + (-3) equals zero.

Similarly, if x is negative, we can choose y to be its positive counterpart. For example, if x = -5, then y = 5. The sum x + y = (-5) + 5 equals zero.

If x is zero, we can simply choose y to be zero as well. The sum x + y = 0 + 0 equals zero.

In each case, we have found a suitable value for y such that x + y equals zero. Therefore, for every integer value of x, there exists an integer value of y such that x + y = 0. Thus, the statement ∀x∃ y:x+y=0 is true for all integers x and y.

2. $\exists y \, \forall x : x + y = x$

Equivalent English Statement: "There exists a value of y such that for every value of x, the sum of x and y equals x."

Claim: The statement is true if and only if y = 0.

Argument:

Let's break down the statement:

Name: Hang Zhao Homework 4 - Analysis

1. "There exists a value of y": This means we need to find at least one value of y that satisfies the given condition.

2. "such that for every value of x, the sum of x and y equals x": This means that regardless of the value of x, the sum of x and y should be equal to x.

Now, let's analyze the conditions:

- If y = 0, then for every integer value of x, x + y = x + 0 = x. This holds true for all integers x, satisfying the given condition.

- If y ≠ 0, then for some integer values of x, the sum x + y will not be equal to x. For example, if y = 1, then for x = 1, we have 1 + 1 ≠ 1. Therefore, the condition doesn't hold true for all integer values of x.

Therefore, the statement is true if and only if y = 0. If y = 0, then for every integer value of x, x + y = x. Otherwise, the statement is false.

3. $\exists x \, \forall y : x + y = x$

Equivalent English Statement: "There exists a value of x such that for every value of y, the sum of x and y equals x."

Claim: The statement is true if and only if y = 0.

Argument:

Let's analyze the statement:

1. "There exists a value of x": This means we need to find at least one value of x that satisfies the given condition.

2. "such that for every value of y, the sum of x and y equals x": This means that regardless of the value of y, the sum of x and y should be equal to x.

Now, let's consider the conditions:

- If y = 0, then for every integer value of x, x + y = x + 0 = x. This holds true for all integers x, satisfying the given condition.

- If y ≠ 0, then for some integer values of x, the sum x + y will not be equal to x. For example, if y = 1, then for any integer value of x, x + 1 ≠ x. Therefore, the condition doesn't hold true for all integer values of x.

Therefore, the statement is true if and only if y = 0. If y = 0, then for every integer value of x, x + y = x. Otherwise, the statement is false.

Name: Hang Zhao Homework 4 - Analysis

Problem 2: Growth of Functions

Organize the following functions into six (6) columns. Items in the same column should have the same asymptotic growth rates (they are big-Oh and big-θ of each other. If a column is to the left of another column, all of its growth rates should be slower than those of the column to its right.

$$n^2,\ n!,\ n\,log_2\,n,\ 3n,\ 5n^2+3,\ 2^n,\ 10000,\ n\,log_3\,n,\ 100,\ 100n$$

Constant Time (O(1))	Linear Time (O(n))	Linearithmic Time (O(n log n))	Quadratic Time ($O(n^2)$)	Exponential Time ($O(2^n)$)	Factorial O(n!)
10000	100n	$nlog_2n$	n^2	2^n	n!
100	3n	$nlog_3n$	$5n^2 + 3$		

Problem 3: Function Growth Language

Match the following English explanations to the *best* corresponding big-Oh function by drawing a line from an element in the left column to an element in the right column.

Constant	$O(n^3)$
Logarithmic	$O(1)$
Linear	$O(n)$
Quadratic	$O(log_2\,n)$
Cubic	$O(n^2)$
Exponential	$O(n!)$
Factorial	$O(2^n)$

Name: Hang Zhao Homework 4 - Analysis

Constant	$O(n^3)$
Logarithmic	$O(1)$
Linear	$O(n)$
Quadratic	$O(log_2 n)$
Cubic	$O(n^2)$
Exponential	$O(n!)$
Factorial	$O(2^n)$

Problem 4: Big-Oh

1. Using the definition of big-Oh, show that $100n + 5 \in O(2n)$

First, we need to find constants c>0 and n_0 >0, therefore for all n>=n_0, 100n + 5<= c*2n.

Choose c = 55 and n_0 = 1. for n>=n_0=1, we have 100n+5<=105n

At this point we need to show that 105n<=55*2n for all n>=n_0

for n>=n_0=1, it is evident that 105n<=55*2n=110n

Thus, 100n+5∈O(2n) with c =55 and n_0 =1. This means that for n >=1, the function 100n+5 is bounded above by 55*2n,verifying its membership in O(2n).

2. Using the definition of big-Oh, show that $n^3 + n^2 + n + 42 \in O(n^3)$

By the definition of the big O symbol，to prove $n^3 + n^2 + n + 42 \in O(n^3)$，We need to find two normal numbers c and n_0, for $n \geq n_0$，all have $n^3 + n^2 + n + 42 \leq c * n^3$ established.

We can break $n^3 + n^2 + n + 42$ down to $(1 + 1/n + 1/n^2 + 42/n^3) * n^3$。When n is large enough，1/n、$1/n^2$ and $42/n^3$ Both are negligible, so we can take c = 45 and n_0 = 1. So for all $n \geq 1$，we have $n^3 + n^2 + n + 42 \leq 45n^3$，that is $n^3 + n^2 + n + 42 \in O(n^3)$.

3. Using the definition of big-Oh, show that $n^{42} + 1{,}000{,}000 \in O(n^{42})$

To show that n^{42} + 1000000 ∈O(n^{42}), we need to find constants c and n^0 such that

$|n^{42} + 1000000| <= c|n^{42}|$, for all n>=$n_0$. Since $|n^{42} + 1000000| <= |n^{42}| + 1000000$ for all n>=1, we can choose c=2 and n_0=1 to show that n^{42} + 1000000 ∈O(n^{42}). When n is large enough, 1000000<=n^{42} ,

So $n^{42} + 1000000 <= n^{42} + n^{42} = 2n^{42}$, so $n^{42} + 1000000 \in O(n^{42})$.

Problem 5: Searching

In this problem, we consider the problem of searching in ordered and unordered arrays:

1. We are given an algorithm called *search* that can tell us *true* or *false* in one step per search query if we have found our desired element in an unordered array of length 2048. How many steps does it takes in the worst possible case to search for a given element in the unordered array?

   In the worst-case scenario, when searching for an element in an unordered array of length 2048 using the search algorithm that can tell us true or false in one step per search query, the maximum number of steps it would take is 2048 steps. This is because, in the worst case, the desired element could be the last element in the array, and we would need to check each element one by one until we reach the end of the array.

2. Describe a *fasterSearch* algorithm to search for an element in an ordered array. In your explanation, include the time complexity using big-Oh notation and draw or otherwise clearly explain why this algorithm is able to run faster.

   In an ordered array, a faster algorithm to search for an element is the Binary Search algorithm. Binary Search works by dividing the array into two halves, comparing the middle element with the target element, determining whether the target element is in the left or right half, and continuing the search in the corresponding subarray until the target element is found or determined to be absent.

   The time complexity of the Binary Search algorithm is O(log n), where n is the length of the array. Compared to linear search, Binary Search can quickly narrow down the search range and eliminate half of the elements at each step. Therefore, when searching for an element in an ordered array, Binary Search is more efficient than linear search.

   Here are the basic steps of the Binary Search algorithm:
   1. Initialize the left and right boundaries, which are the start and end positions of the array.
   2. Calculate the index of the middle element within the current search range.
   3. Check if the middle element is equal to the target element, if so, return.
   4. If the middle element is greater than the target element, update the right boundary to be one position before the middle element.
   5. If the middle element is less than the target element, update the left boundary to be one position after the middle element.
   6. Repeat steps 2-5 until the target element is found or determined to be absent.

   By following these steps, the Binary Search algorithm can efficiently search for a target element in an ordered array with a time complexity of O(log n).

3. How many steps does your *fasterSearch* algorithm (from the previous part) take to find an element in an ordered array of length 2,097,152 in the worst case? Show the math to support your claim.

To find an element in an ordered array of length 2,097,152 using the Binary Search algorithm in the worst-case scenario, we can calculate the number of steps it would take based on the logarithmic nature of the Binary Search algorithm.

Given that the length of the array is 2,097,152, which is 2^{21} (since 2^{21} = 2,097,152), in the worst-case scenario, Binary Search would require $\log_2(2^{21})$ comparisons to find the element.

$\log_2(2^{21}) = 21$

Therefore, in the worst-case scenario, the Binary Search algorithm would take 21 steps to find an element in an ordered array of length 2,097,152.

Problem 6: Another Search Analysis

Imagine it is your lucky day, and you are given 100 golden coins. Unfortunately, 99 of the gold coins are fake. The fake gold coins all weight 1 oz. but the real gold weighs 1.0000001 oz. You are also given one balancing scale that can precisely weight each of the two sides. If one side is heavier than the other the other side, you will see the scale tip.

1. Describe an algorithm for finding the real coin. You must also include the algorithm's time complexity. **Hint:** Think carefully – or do this experiment with a roommate and think about how many ways you can prune the maximum number of fake coins using your scale.

To find the real gold coin among the 100 coins, you can follow this algorithm:

1. Divide the 100 coins into two groups of 50 coins each.

2. Weigh one group of 50 coins against the other group on the balancing scale.

3. If one side is heavier, it means the real coin is in that group of 50 coins. If they balance, the real coin is in the other group of 50 coins.

4. Take the group of 50 coins that contains the real coin and repeat the process by dividing them into two groups of 25 coins each.

5. Weigh one group of 25 coins against the other group.

6. Continue this process of dividing and weighing until you are left with only one coin on the scale.

The time complexity of this algorithm is $O(\log_2 100) = O(7)$, which means you would need at most 7 weighings to find the real gold coin among the 100 coins. This algorithm efficiently prunes the maximum number of fake coins at each step, allowing you to identify the real coin with a minimal number of weighings.

2. How many weightings must you do to find the real coin given your algorithm?

According to the algorithm described, you would need to do a maximum of 7 weighings to find the real gold coin among the 100 coins. Each weighing helps you divide the coins into two groups, narrowing down the possibilities until you are left with only one coin, which is the real gold coin.

Problem 7 – Insertion Sort

1. Explain what you think the worst case, big-Oh complexity and the best-case, big-Oh complexity of insertion sort is. Why do you think that?

Insertion Sort is a simple sorting algorithm that builds the final sorted array one element at a time. Here's an explanation of the worst-case and best-case scenarios along with their corresponding big-O complexities for Insertion Sort:

1. Worst-case scenario:

- In the worst-case scenario, when the input array is in reverse order, each element needs to be compared and moved to its correct position in the sorted part of the array.

- The worst-case time complexity of Insertion Sort is O(n^2), where n is the number of elements in the array.

- This worst-case complexity arises when each element in the unsorted part of the array needs to be compared and moved to its correct position in the sorted part.

2. Best-case scenario:

- In the best-case scenario, when the input array is already sorted, Insertion Sort only requires comparisons to verify that each element is already in its correct position.

- The best-case time complexity of Insertion Sort is O(n), where n is the number of elements in the array.

- This best-case complexity occurs when the array is already sorted, and Insertion Sort only needs to check each element once to confirm that it's in the right place.

In summary, Insertion Sort has a worst-case time complexity of O(n^2) and a best-case time complexity of O(n). The worst-case complexity arises from having to move each element to its correct position, while the best-case complexity occurs when the array is already sorted, requiring minimal comparisons to verify the order.

2. Do you think that you could have gotten a better big-Oh complexity if you had been able to use additional storage (i.e., your implementation was not *in-place*)?

If the Insertion Sort algorithm were allowed to use additional storage (i.e., not an in-place sorting algorithm), it could potentially achieve a better big-O complexity in certain scenarios.

Name: Hang Zhao Homework 4 - Analysis

By using additional storage, a more efficient sorting algorithm like Merge Sort or Quick Sort could be implemented, which have better average-case time complexities compared to Insertion Sort. These algorithms can achieve O(n log n) time complexity on average, which is more efficient than the O(n^2) worst-case complexity of Insertion Sort.

Therefore, using additional storage and implementing a different sorting algorithm could lead to improved time complexity in certain cases compared to the in-place Insertion Sort algorithm.