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

Answer:

For all integers x, there exists an integer y such that x+y=0.This statement means that given any integer x, there is always another integer y that can be added to it to get a sum of zero.

This statement is true for all integers x.

Proof:

Let y = -x. Then x + y = x + (-x) = 0, which satisfies the equation. Since x can be any integer, this holds true for all integers.

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

Answer:

There exists an integer y such that for all integers x, x+y=x. This statement means that there is some integer y such that adding it to any integer x does not change the value of x.

This statement is true for all integers y.

Proof:

Let y be any integer. Then for any integer x, x+y=x+(y-0)=x+y, which satisfies the equation. Since y can be any integer, this holds true for all integers.

Name: Chaoyi Jiang Homework 4 - Analysis

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

Answer:

There exists an integer x such that for all integers y, x+y=x. This statement means that there is some integer x such that adding any integer y to it does not change the value of x.

This statement is false for all integers x.

Proof:

Assume that there exists an integer x such that x+y=x for all integers y. Then let y=1, which means that x+1=x. Subtracting x from both sides gives 1=0, which is a contradiction. Therefore, there is no integer x that satisfies the statement for all integers y.

Name: Chaoyi Jiang 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.

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

Answer:

Column 1	Column 2	Column 3	Column 4	Column 5	Column 6
100	n	3n	100n	n 〖log〗_2 n	10000
				n 〖log〗_3 n	n^2
					5n^2+3
					n!
					2^n

Explanation:

Column 1 contains the constant time functions 100 and 10000.

Column 2 contains linear time functions n and 100n.

Column 3 contains functions that grow linearly with a small constant factor, namely 3n.

Column 4 contains functions that grow slower than n 〖log〗_2n but faster than linear time, namely n 〖log〗_3n.

Column 5 contains quadratic time function n^2 and n 〖log〗_2n which grows slower than the remaining functions in column 6 but faster than those in columns 1-4.

Column 6 contains the exponential function 2^n, and the polynomial function 5n^2+3, which grow faster than all the other functions. The factorial function n! grows even faster than these functions and is placed in the same column as 5n^2+3 and 2^n.

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: Chaoyi Jiang Homework 4 - Analysis

Answer:

Constant ----------> O(1)

Logarithmic ----------> $O(\log_2 n)$

Linear ----------> O(n)

Quadratic ----------> $O(n^2)$

Cubic ----------> $O(n^3)$

Exponential --------> $O(2^n)$

Factorial ----------> O(n!)

Problem 4: Big-Oh

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

Answer:

For f(n) = 100n + 5 and g(n) = 2n:

We have f(n) = 100n + 5 ≤ 100n + 100n = 200n for all n ≥ 1.

Let c = 200 and n0 = 1, then 100n+5 ≤ c*2n for all n ≥ n0. Therefore, 100n+5 ∈ O(2n).

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

Answer:

For f(n) = n^3 + n^2 + n + 42 and g(n) = n^3:

We have f(n) = n^3 + n^2 + n + 42 ≤ n^3 + n^3 + n^3 + n^3 = 4n^3 for all n ≥ 1.

Let c = 4 and n0 = 1, then n^3 + n^2 + n + 42 ≤ c*n^3 for all n ≥ n0. Therefore, n^3 + n^2 + n + 42 ∈ O(n^3).

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

Name: Chaoyi Jiang Homework 4 - Analysis

For $f(n) = n^{42} + 1{,}000{,}000$ and $g(n) = n^{42}$:

We have $f(n) = n^{42} + 1{,}000{,}000 \leq 2n^{42}$ for all $n \geq 1$, since n^{42} is a dominant term.

Let $c = 2$ and $n_0 = 1$, then $n^{42} + 1{,}000{,}000 \leq c*n^{42}$ for all $n \geq n_0$. Therefore, $n^{42} + 1{,}000{,}000 \in O(n^{42})$.

Name: Chaoyi Jiang Homework 4 - Analysis

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?

Answer:

In the worst case, the element being searched for may be the last one in the array, so we would need to search through all 2048 elements. Therefore, the worst-case scenario for the search algorithm would take 2048 steps.

2. Describe a *faster Search* 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.

Answer:

The faster Search algorithm for an ordered array works by repeatedly dividing the array in half until the desired element is found. Here is how it works:

Start by setting two pointers: low and high. low points to the first element in the array and high points to the last element in the array.

Compute the middle index of the array as the average of low and high: mid = (low + high) / 2.

Check if the middle element of the array is equal to the desired element. If it is, return true.

If the middle element is greater than the desired element, set high to mid - 1 (i.e., search the left half of the array). Otherwise, set low to mid + 1 (i.e., search the right half of the array).

Repeat steps 2-4 until the desired element is found or low is greater than high (in which case the element is not in the array).

The time complexity of this algorithm is O(log n), where n is the length of the array. This is because at each step, the size of the search space is halved. This means that even for very large arrays, the algorithm can find the desired element in a relatively small number of steps.

The reason why this algorithm is faster than the search algorithm for an unordered array is because it takes advantage of the fact that the array is ordered. By repeatedly dividing the array in half, we can eliminate large portions of the search space with each step, allowing us to find the desired element more quickly.

3. How many steps does your *faster Search* 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.

Answer:

If we have an ordered array of length 2,097,152, then the number of steps it takes for the faster Search algorithm to find an element in the worst case is log2(2,097,152) = 21. This is because log2(n) is the number of times we can divide n in half before reaching 1. Therefore, the faster Search algorithm can find an element in an ordered array of length 2,097,152 in at most 21 steps in the worst case.

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.

Answer:

Algorithm:

To find the real coin, we can divide the 100 coins into two groups of 50 each. We can then weigh these two groups against each other on the balancing scale. If the two groups are of equal weight, then we know that the real coin is in the remaining 50 coins. We can repeat this process, dividing the remaining 50 coins into two groups of 25 and weighing them against each other. If they are of equal weight, we know the real coin is in the remaining 25 coins. We can keep repeating this process, dividing the remaining coins in half and weighing them against each other until we find the real coin.

If at any point the two groups on the balancing scale are not of equal weight, then we know that the real coin must be in the group that weighs more. We can repeat the process with the heavier group until we find the real coin.

The time complexity of this algorithm is O(log2(n)) as we are dividing the number of coins in half at each step until we find the real coin.

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

Answer:

We will need a maximum of 7 weightings to find the real coin using this algorithm. Here is how it works:

- First weighing: Weigh 50 coins against the other 50 coins. If they are of equal weight, we know the real coin is in the remaining 50 coins. We have eliminated 50 fake coins.
- Second weighing: Divide the remaining 50 coins into two groups of 25 and weigh them against each other. If they are of equal weight, we know the real coin is in the remaining 25 coins. We have eliminated 75 fake coins.
- Third weighing: Divide the remaining 25 coins into two groups of 12 and weigh them against each other. If they are of equal weight, we know the real coin is in the remaining 13 coins. We have eliminated 87 fake coins.
- Fourth weighing: Divide the remaining 13 coins into two groups of 6 and weigh them against each other. If they are of equal weight, we know the real coin is in the remaining 7 coins. We have eliminated 93 fake coins.
- Fifth weighing: Divide the remaining 7 coins into two groups of 3 and weigh them against each other. If they are of equal weight, we know the real coin is in the remaining 1 coin. We have eliminated 97 fake coins.
- Sixth weighing: Weigh one of the 3 coins against another. If they are of equal weight, we know the remaining coin is the real one. We have eliminated 98 fake coins.
- Seventh weighing: If the two coins are of different weights, then the real coin is the heavier one. We have eliminated 99 fake coins.

Name: Chaoyi Jiang Homework 4 - Analysis

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?

Answer:

The worst-case big-Oh complexity of insertion sort is O(n^2), where n is the number of elements in the array being sorted. This occurs when the array is in reverse order, and every element must be shifted over for each comparison. The best-case big-Oh complexity is O(n), which occurs when the array is already sorted or nearly sorted, and only a few comparisons and shifts are needed.

I think that the worst-case complexity is O(n^2) because in the worst case scenario, each element needs to be compared and shifted all the way to the beginning of the array. This requires n comparisons and potentially n shifts for each element, resulting in a quadratic time complexity.

The best-case complexity of O(n) is achievable because in the best case scenario, the array is already sorted or nearly sorted, and each element only needs to be compared to a few other elements before finding its correct position.

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*)?

Answer:

Yes, if additional storage was available, it is possible to achieve a better big-Oh complexity for insertion sort. By using additional storage, such as a separate array or linked list, the algorithm can avoid the need to shift elements over to make room for a new element. This would reduce the number of operations needed and result in a faster algorithm. However, using additional storage would come with a trade-off of increased memory usage and potential overhead, which may not be worth it depending on the specific use case.

Name: Chaoyi Jiang Homework 4 - Analysis