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

English translation: For all integers x, there exists an integer y such that $x + y = 0$.

Proof True.Let x be an arbitrary integer. Define y to be the additive inverse of x, which is $-x$. By the definition of additive inverses in the set of integers, we have: $\mathrm{x} + (\text{-x}) = 0$ Since x is an arbitrary integer and the set of integers is closed under negation, $-x$ is also an integer. Therefore, for every integer x, there exists an integer $y = -x$ such that their sum is zero.

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

English translation: There exists an integer y such that for every integer x, $x + y = x$.

Proof True.We want to show the existence of an integer y which, when added to any integer x, results in x. Let us choose $y = 0$, the additive identity in the set of integers. By the property of additive identity, for any integer x, we have: $\mathrm{x} + 0 = \mathrm{x}$ Thus, there exists an integer $y = 0$ such that for every integer x, the sum of x and y equals x.

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

English translation: There exists an integer x such that for every integer y, $x + y = x$.

Proof False.Assume, to the contrary, that such an integer x exists. Then, for any integer y, we must have $x + y = x$.

Let y be an integer such that $y \neq 0$ (for instance, $y = 1$). According to our assumption, $x + y = x$ should hold true. However, adding y to both sides of the equation $x = x$ would give us $x + y = x + 0$.

By the cancellation law, we can subtract x from both sides of the equation to obtain $y = 0$. This is a contradiction since we have chosen y to be a non-zero integer. Therefore, our initial assumption that such an integer x exists is false.

Name: Rong Huang Homework 4 - Analysis

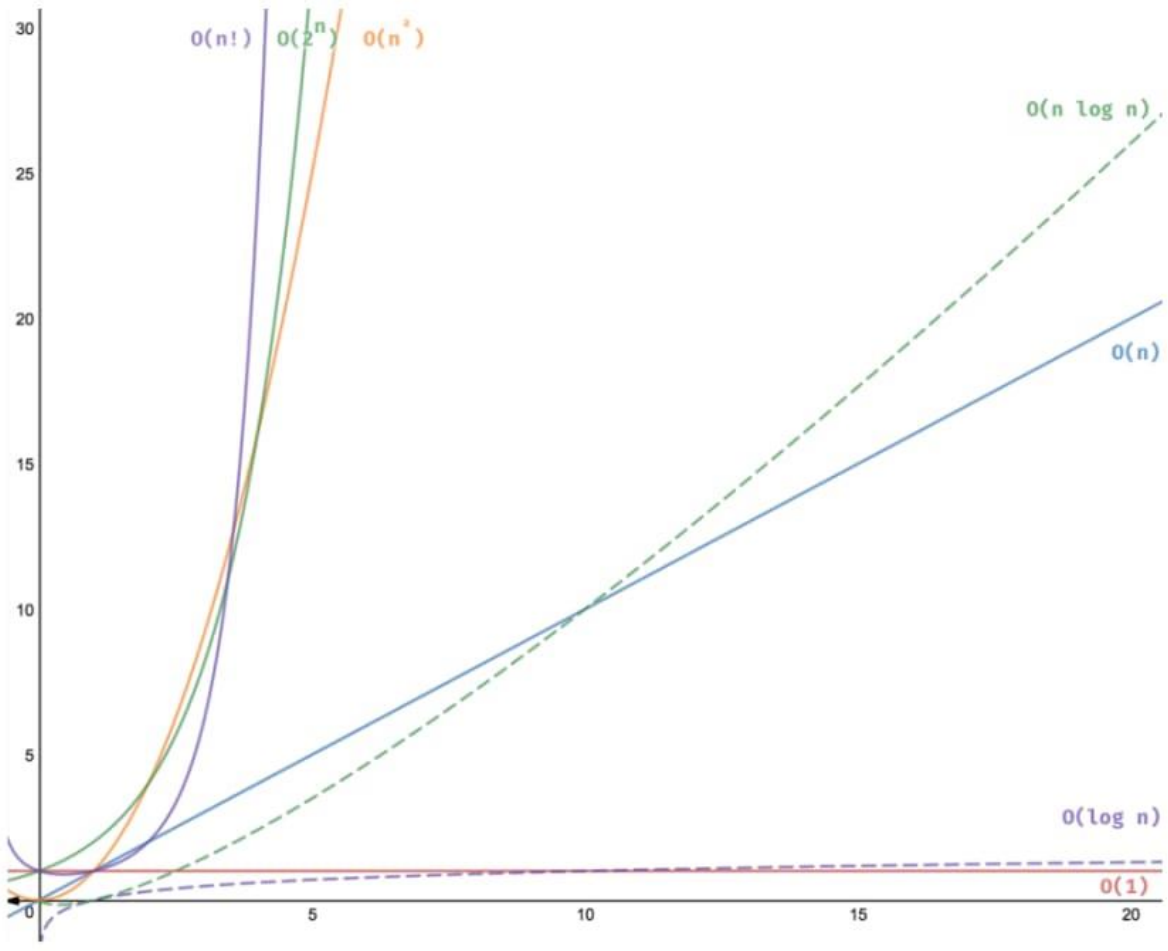
Problem 2: Growth of Functions

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: $O(nlogn)$	Quadratic Time: $O(n^2)$	Exponential Time:$O(2^n)$	Factorial Time:$O(n!)$
100	3n	$n\log_2\ n$	n^2	2^n	$n!$
10000	100n	$n\log_3\ n$	$5n^2+3$		

And the picture is just like this:

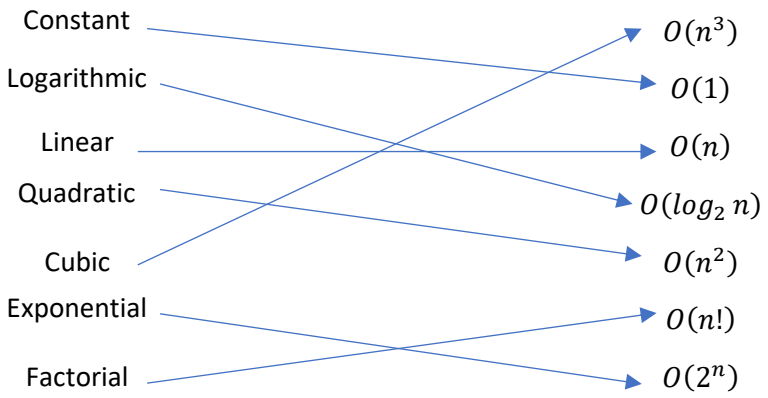


Cite: https://dev.to/bks1242/time-complexity-5c3

Name: Rong Huang Homework 4 - Analysis

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.



Problem 4: Big-Oh

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

Definition: To show that $f(n)$ is $O(g(n))$, we need to find constants $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$.

Proof:

Let $f(n) = 100n + 5$ and $g(n) = 2n$. We aim to show that $f(n)$ is $O(g(n))$, meaning there exists constants $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0$, $100n + 5 \leq c \cdot 2n$.

Choose $c = 51$. Then for $c \cdot g(n) = 51 \cdot 2n = 102n$.

Now, we need to find n_0 such that $100n + 5 \leq 102n$ for all $n \geq n_0$.

Notice that for $n = 1$, $100 \cdot 1 + 5 = 105 \leq 102 \cdot 1 = 102$, which does not satisfy the inequality. However, as n increases, the additional constant 5 becomes negligible.

Let's try $n_0 = 3$, for $n \geq 3$, $100n + 5$ becomes less significant compared to $102n$ because the difference $2n - 5$ becomes positive and increases with n.

Thus, for $n \geq 3$, $100n + 5 \leq 102n$ is satisfied.

Therefore, we have shown that there exists constants $c = 51$ and $n_0 = 3$ such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$, which means $100n + 5$ is $O(2n)$.

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

Definition: To show that $f(n)$ is $O(g(n))$, we need to find constants $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$.

Proof:

Let $f(n) = n^3 + n^2 + n + 42$ and $g(n) = n^3$. We aim to show that $f(n)$ is $O(g(n))$, meaning there exists constants $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0$, $n^3 + n^2 + n + 42 \leq c \cdot n^3$.

Observe that for $n \geq 1$, $n^2 \leq n^3$, $n \leq n^3$, and $42 \leq n^3$ because each term on the left can be less than or equal to n^3 for sufficiently large n.

Adding these inequalities, we get $n^3 + n^2 + n + 42 \leq n^3 + n^3 + n^3 + n^3 = 4n^3$.

Therefore, we can choose $c = 4$ and $n_0 = 1$ to satisfy the definition of big-Oh notation. For all $n \geq n_0$, $f(n) \leq c \cdot g(n)$ is true, which means $n^3 + n^2 + n + 42$ is $O(n^3)$.

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

Definition: To show that $f(n)$ is $O(g(n))$, we need to find constants $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$.

Proof:

Consider $f(n) = n^{42} + 1,000,000$ and $g(n) = n^{42}$. To prove $f(n)$ is $O(g(n))$, we need to demonstrate the existence of constants $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0$, $n^{42} + 1,000,000 \leq c \cdot n^{42}$.

Select $c = 2$. The inequality $n^{42}+1,000,000 \leq 2 \cdot n^{42}$ simplifies to $1,000,000 \leq n^{42}$, which holds true for all n large enough, since n^{42} grows much faster than a constant term.

To identify n_0, observe that any n sufficiently large to ensure $n^{42} > 1,000,000$ will satisfy the condition. For example, choosing n_0 such that $n_0^{42} > 1,000,000$ will guarantee that $n^{42} + 1,000,000 \leq 2n^{42}$ for all $n \geq n_0$.

Hence, with $c = 2$ and an appropriate n_0, we've shown that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$, proving $n^{42} + 1,000,000$ is $O(n^{42})$.

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?

   For an unordered array, the only way to find an element is to check each element one by one until we find the desired element or conclude that it's not present. This is known as linear search.

   In the worst case, the element we are looking for is either at the end of the array or not present in the array at all. Therefore, we would have to check every single element.

   So, if the array length is 2048, in the worst case, we would need to check all 2048 elements, which means the algorithm would take 2048 steps.

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.

   When an array is ordered, we can use a more efficient algorithm called binary search. Binary search starts by comparing the middle element of the array to the target value. If the target value matches the middle element, the search is complete. If the target value is less or greater (depending on the order of sorting), the search continues on the sub-array to the left or the right of the middle element, respectively. This process is repeated on the sub-array, each time dividing the remaining portion of the array in half, which makes the search much faster.

   The time complexity of binary search is O(log n), where n is the number of elements in the array. This is because with each comparison, we halve the number of elements we are looking at, which is a logarithmic reduction. Imagine a phone book, instead of searching for a name by starting from the first page, you open it to the middle and see if the name is before or after the page you opened. Then, you split the relevant half again and repeat the process. This is exactly how binary search works.

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.

In the worst case, binary search takes the base-2 logarithm of the array length because it halves the array at each step.

The worst-case number of steps is given by:

$$\text{steps} = \lceil \log_2(n) \rceil$$

Where *n* is the length of the array. Let's calculate this for an array length of 2,097,152.

It would take 21 steps in the worst case to find an element in an ordered array of length 2,097,152. The math behind this is the calculation of the base-2 logarithm of the array length, rounded up to the nearest whole number.

Name: Rong Huang Homework 4 - Analysis

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.

The algorithm to apply for this is essentially a binary search. Each weighing halves the number of possible coins that could be real.

1) Divide the coins into two groups of 50 coins each. Weigh the two groups against each other.
2) Take the heavier group (since the real coin is slightly heavier, the group containing the real coin will be heavier) and divide it into two groups of 25 coins each. Weigh these two groups against each other.
3) Continue this process of dividing the group in half and weighing the two subgroups against each other. Each time, keep the heavier group for the next division.
4) When you reach a group size of 1, that coin is the real gold coin.

The time complexity of this algorithm is O(log n), where n is the number of coins.

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

Since we're dividing the group of coins in half each time and starting with 100 coins, we'll use the base-2 logarithm to find the number of steps, rounding up to the nearest whole number since we can't do a fraction of a weighing.

Using the algorithm described, we would need to perform 7 weightings to find the real gold coin among the 100 coins.

Name: Rong Huang 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?

Worst Case Complexity: O(n^2)

In the worst case, insertion sort runs in O(n^2) time complexity. This happens when the array is sorted in reverse order. Each element has to be compared with all the other elements that have already been sorted (to its left), resulting in about $\frac{n(n-1)}{2}$ comparisons and swaps, which simplifies to O(n^2).

Best Case Complexity: O(n)

In the best case, insertion sort runs in O(n) time complexity. This occurs when the array is already sorted. Each element only needs to be compared once with the element before it; since it's already in the correct position, no swaps are needed. So, there are n-1 comparisons and 0 swaps, which is linear time, O(n).

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*)?

Using additional storage, as seen with Merge Sort, facilitates a more efficient sorting process with a time complexity of O(n log n). This is faster than the O(n^2) complexity of Insertion Sort because Merge Sort organizes and combines smaller sorted sections of the array, leveraging the additional storage to keep track of these operations. This method is more efficient, especially for larger arrays, since it significantly reduces the number of comparisons and swaps needed to sort the entire array.