# 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 and are integers (e.g., they can be any integer). Then write a convincing argument to prove your claim.

For all x, there exists a y such that x plus y equals zero

We can use proof by construction to prove this:

We can choose y to be -x: x+y=x+(-x)=0. This is true for any integer x, so we can prove that for all x, there exists a y that x plus y equals zero.

There exists a y such that for all x, x plus y equals x.

We can use direct proof to prove this claim: we can choose y to be 0, such that for any integer x, x+y=x+0=x is true

There exists an x such that for all y, x plus y equals x.

To prove this, suppose x =1, then for any integer y is not equals to 0, then we have 1+y is not equals to 1.

Therefore, there is no integer x that satisfies this statement.

## 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- 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.

, , , , , , , , ,

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 10000 | 100 | n | 3n |  |  |
|  |  | , |  |  |  |

## 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 |  |
| Logarithmic |  |
| Linear |  |
| Quadratic |  |
| Cubic |  |
| Exponential |  |
| Factorial |  |

## Problem 4: Big-Oh

1. Using the definition of big-Oh, show that

|100n +5| <= c\*|2n| for all n>=n0

Choose c =105 and n0=1,

Then for all n >=1, we have |100n +5| =100n+5 <= 105\*2n =c\*|2n|

Then 100n +5 <= 105 \*2n

50n +2.5 <=52.5 \*2n

n + 0.05 <= 2.1\* 2n

then choose n0=1, for all n>=1, then we have:

2.1 \* 2n >= 2.1 \*2^1=4.2

Then we have n+ 0.05 <=2.1 \*2n

1+0.05<=2.1\*2^1

1.05<=4.2

So there exist constants c=105 and n0=1 such that |100n +5| <= c\*|2n| for all n>=n0

1. Using the definition of big-Oh, show that

Choose c=3 and n0=1. <

For all n>=1, we have |=<=3\*=c\*|

Then

Choose n0-1, then for all n>=1, we have 2

>=21 (not true)

>=21 (not true)

=27 >=21 (true)

Thus, we have shown that exist constants c=3 and n0=1 such that

| c\*| for all n>=n0

1. Using the definition of big-Oh, show that

|

Choose c=2 and n0=1. Then for all n>=1, we have

| +1000000<=2\*

Then +1000000 <=2

Choose n0=1, then for all n>=1, we have 2\*

1^42 <500000 (not true)

2^42 <500000(not true)

Thus we can show that there exist constants c =2 and n0 =1 such that

|+1000000|<=c\*|

This means that +1000000

## 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?

The worst scenario is that the element we are searching for is the last element of the array or it is not in the array, so it will take 2048 steps to search for a given element in an unordered array of length 2048 using the search algorithm.

1. 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.

Binary search:

1. First set two pointers: left and right, to the first and last element of the array.
2. Second step calculate the mid-point as (left+right)/2
3. Compare the middle element. Return true if they are equal.
4. If the element we are searching for is less than the middle element, we set the right pointer to the middle index-1.
5. If the element we are searching is greater than the middle element, we set the left pointer to the middle index+1
6. Then repeat step 2
7. Continue these steps until we find the element we are searching for.

Binary search is faster than linear search is because it always decrease the range of searching by eliminating a half of the elements for each comparison.

1. 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.

The worst case is when the element we are searching for is at the first or last of the array, or it is not in the array.

Log2(2097152) = 21

So it will take 21 steps

## 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.

It is suitable to use binary search to solve this problem.

We can divide the 100 coins into 50 and 50 then weight two groups, the real coin is in the heavier group.

The repeatively divide the heavier group in to two equal number groups, and the real coin will always be in the heavier group.

The time complexity of this algorithm is O(log n).

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

We can find the real coin with at most 7 weighings.

Log2(100) = 7

## 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?

The worst case time complexity of insertion sort is O(n^2), where n is the number of elements to be sorted. This case each element has to be compared to all the elements before it, and it is shifted to the correct position. The best case time complexity of insertion sort is O(n). In this case, each element is compared to the one before it but no shifting is needed. The average time complexity id O(n^2).

The best case complexity if O(n) is because the inner loop only needs to make one comparison for each element. The worst case complexity is O(n^2) is because the inner loop has to iterate through all elements before finding the correct one, and the number of comparisons and shifts requires grows exponentially with the number of elements.

1. 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*)?

Yes, we might achieve a better big-O complexity if additional storage is allowed, we can achieve the worst case complexity of O(nlogn) for merge sort, and it is faster than the worst case complexity of O(n^2) for insertion sort.