CS-GY 6033 : Design and Analysis of Algorithms , Section C : Assignment-1

Khwaab Thareja — N15911999 — kt3180

September 2024

1 Asymptotic Notation Comparison

Take the following list of functions and arrange them in ascending order of growth rates. That is, if function $g(n)$ immediately follows function $f(n)$ in your list, then it should be the case that $f(n)$ is $O(g(n))$. You do not need to justify your answer for this problem part.

- $f_1(n) = \sqrt{2}n$
- $f_2(n) = 2^{\sqrt{\log n}}$
- $f_3(n) = \log n$
- $f_4(n) = 100n$
- $f_5(n) = n^2 \log n$
- $f_6(n) = n^{1/\log\log n}$
- $f_7(n) = n!$
- $f_8(n) = 2^n \cdot n!$
- $f_9(n) = n^{0.99n}$
- $f_{10}(n) = n^n$

Recall that $\log(n!) > 0.99n \log n$ for sufficiently large n.

Answer 1

$\log(n) < 2^{\sqrt{\log(n)}} < n^{1/\log\log(n)} < \sqrt{2}n < 100n < n^2\log(n) < n^{0.99n} < n! < 2^n{\cdot}n! < n^n$

$f_3(n) < f_2(n) < f_6(n) < f_1(n) < f_4(n) < f_5(n) < f_9(n) < f_7(n) < f_8(n) < f_{10}(n)$

2. Asymptotic Notation Properties

Assume both f, g, h, and k are asymptotically positive functions. Decide if each of the following statements is true or false. If it is true, prove it; if it is false, give a counterexample.

(a) Does $f(n) = O(g(n))$ and $g(n) = O(f(n))$ imply that $f(n) = \Theta(g(n))$? Prove or disprove.

Answer (a):

True

From $f(n) = O(g(n))$, we know that

$$f(n) \leq c_1 \cdot g(n) \quad \text{where} \quad c_1 > 0 \text{ and } n \geq n_1 \tag{I}$$

Similarly, we have $g(n) = O(f(n))$. Therefore,

$$g(n) \leq c_2 \cdot f(n) \quad \text{where} \quad c_2 > 0 \text{ and } n \geq n_2 \tag{II}$$

Combining (I) and (II) and taking $f(n)$ common, we get:

$$\frac{1}{c_2} \cdot g(n) \leq f(n) \leq c_1 \cdot g(n)$$

Here, $\frac{1}{c_2}$ is some constant which we can denote as c_3.
Therefore, we have:

$$c_3 \cdot g(n) \leq f(n) \leq c_1 \cdot g(n)$$

Thus, we conclude $f(n) = \Theta(g(n))$.
Let $f(n) = 3n^2 + 2$. To show that $f(n) = O(g(n))$, we need to satisfy the condition:

$$3n + 2 \leq c_1 \cdot g(n)$$

for large n. Therefore, we can ignore the constant 2. Hence, we have:

$$3n + 2 \leq c_1 \cdot 6n \quad \text{for large } n,$$

which simplifies to:

$$3n \leq c_1 \cdot 6n.$$

Thus, we find $c_1 \geq 1$.

Similarly, for $g(n) = O(f(n))$:

$$g(n) \leq c_2 \cdot f(n) \implies 6n \leq c_2 \cdot (3n + 2).$$

This can be rearranged to show:

$$6n \leq c_2 \cdot 3n + 2c_2.$$

For large n, we can ignore the constant $2c_2$:

$$6n \leq c_2 \cdot 3n.$$

Rearranging gives:

$$c_2 \geq 2.$$

Hence, we get:

$$3n \leq 3n + 2 \leq 6n \quad \text{for } c_1 > 1 \text{ and } c_2 > 2.$$

Thus, we conclude that:

$f(n) = \Theta(g(n))$

Hence the statement is True

(b) Does $f(n) = O(g(n))$ imply $f(n)^2 = O(g(n)^2)$? Prove or disprove.
Answer (b):

True
From $f(n) = O(g(n))$, we know that

$$f(n) \leq c_1 \cdot g(n) \quad \text{where} \quad c_1 > 0 \text{ and } n \geq n_1 \tag{I}$$

Now, if we square both sides of inequality (I), we get:

$$f(n)^2 \leq (c_1^2) \cdot g(n)^2$$

Here, c_1^2 is a constant, and we can rewrite it as c_2. Also, $f(n)^2$ and $g(n)^2$ are still functions of n, but now with a constant c_2.

Therefore, we conclude that:

$$f(n)^2 = O(g(n)^2)$$

Hence, the statement is true.

(c) Does $f(n) = O(g(n))$ and $h(n) = O(k(n))$ imply $f(n) \cdot h(n) = O(g(n) \cdot k(n))$? Prove or disprove.

Answer (c):

True
From $f(n) = O(g(n))$, we know that

$$f(n) \leq c_1 \cdot g(n) \quad \text{where} \quad c_1 > 0 \text{ and } n \geq n_1 \tag{I}$$

Similarly, from $h(n) = O(k(n))$, we have

$$h(n) \leq c_2 \cdot k(n) \quad \text{where} \quad c_2 > 0 \text{ and } n \geq n_2 \tag{II}$$

Multiplying both inequalities (I) and (II), we get:

$$f(n) \cdot h(n) \leq c_1 \cdot g(n) \cdot c_2 \cdot k(n)$$

Thus,

$$f(n) \cdot h(n) \leq c_1 \cdot c_2 \cdot g(n) \cdot k(n)$$

Here, $f(n)$, $h(n)$, $g(n)$, and $k(n)$ are all functions of n and asymptotically positive. The constant $c_1 \cdot c_2$ can be rewritten as c_3. Therefore, we obtain:

$$f(n) \cdot h(n) = O(g(n) \cdot k(n))$$

Hence, the statement is true.

(d) Does $f(n) = O(g(n))$ and $h(n) = O(k(n))$ imply $f(h(n)) = O(g(k(n)))$? Prove or disprove.

We will disprove the statement.

From $f(n) = O(g(n))$, we know that

$$f(n) \leq c_1 \cdot g(n) \quad \text{where} \quad c_1 > 0 \text{ and } n \geq n_1 \tag{I}$$

Similarly, from $h(n) = O(k(n))$, we have

$$h(n) \leq c_2 \cdot k(n) \quad \text{where} \quad c_2 > 0 \text{ and } n \geq n_2 \tag{II}$$

Suppose here that $h(n)$ and $k(n)$ are functions of n with order greater than or equal to 2, i.e., the degree of the equation is ≥ 2. Let us define:

$$h(n) = n^2 \quad \text{and} \quad k(n) = n^2$$

From inequality (II), we have:

$$h(n) \leq c_2 \cdot k(n) \implies n^2 \leq c_2 \cdot n^2$$

Let

$$f(n) = 2^n, \quad g(n) = 2^n, \quad h(n) = n^2, \quad \text{and} \quad k(n) = n^2 - n.$$

So we will have $h(n) \leq c_2 \cdot k(n)$, i.e., $n^2 \leq c_2 \cdot (n^2 - n)$ holds true. However,

$$f(h(n)) = 2^{n^2} \quad \text{and} \quad g(k(n)) = 2^{n^2-n}.$$

Clearly, $f(h(n)) > g(k(n))$ for every constant c.

From inequality (I), there must exist a constant for $f(n) = O(g(n))$ to hold true, but that does not exist since, in this case, $f(h(n)) \geq g(k(n))$ for each constant as $f(n)$ grows exponentially
faster. Hence, the statement is false.

(e) Does $f(n) = O(g(n))$ imply $\log f(n) = O(\log g(n))$? Prove or disprove.

From $f(n) = O(g(n))$, we know that

$$f(n) \leq c_1 \cdot g(n) \quad \text{where } c_1 > 0 \text{ and } n \geq n_1 \quad \text{—- I}$$

Taking the logarithm of both sides, we get:

$$\log(f(n)) \leq \log(c_1 \cdot g(n))$$

Using the properties of logarithms, this can be rewritten as:

$$\log(f(n)) \leq \log(c_1) + \log(g(n))$$

The term $\log(c_1)$ becomes negligible for sufficiently large n and hence we can ignore it.

However, with a counterexample where $f(n) = 2$ and $g(n) = 1 + \frac{1}{n}$, we have:

$$\log(f(n)) = \log(2) \quad \text{and} \quad \log(g(n)) = \log\left(1 + \frac{1}{n}\right).$$

Therefore, $\log(2) > \log(1 + 1/n)$ for any constant $c > 0$. Hence, the condition does not hold.

The statement is therefore **False**.

3. Euclidean Algorithm

Consider the following Euclidean algorithm for computing the greatest common divisor of two integers $a, b > 0$:

```
Algorithm 1 Euclid(a, b)
1: while b > 0 do
2:
3:
4:     t ← b
5:     b ← a \mod b
6:     a ← t
7: end while
8: return a
```

(a) Prove that the algorithm terminates in $O(\log a)$ iterations.

(b) Show that the $O(\log a)$ bound is tight. In other words, prove that there exists a constant $c > 0$ such that for every $n_0 > 0$, there exist two integers a, b such that $a > b > n_0$ and Euclid(a, b) takes at least $c \log a$ iterations to complete.

Answer A

The Euclidean Algorithm to compute GCD clearly suggests that it keeps on going until the lower number becomes 0. Let's take an example and compute the iterations.

Let's consider two numbers: 480 and 150 and find the GCD for these numbers using the Euclidean algorithm. So as per the algorithm, $(480, 150) \rightarrow (150, 480 \mod 150 = 30) \rightarrow (30, 150 \mod 30 = 0)$. The GCD is 30 and we get this in 2 iterations.

Working with the algorithm, each iteration will reduce from (a, b) to $(b, a \mod b)$, where $a \mod b$ will always be less than b. In fact, it is getting reduced to less than half in each iteration.

So, if we take $(a_0, b_0) = (480, 150) \rightarrow (b_0, a_0 \mod b_0) = (a_1, b_1) = (150, 30) \rightarrow (b_1, a_1 \mod b_1) = (a_2, b_2) = (30, 0)$, going in this systematic way for some other numerical values as well, we might find the results of the Euclidean algorithm in n iterations. The last pair would be $(a_n, b_n) = (b_{n-1}, 0)$.

Since it took n iterations for b_0 to become 0, and in each iteration, the value is dropping by approximately half, if b_0 is the starting value, then the last value of b_n before reaching 0 will be $b_0/2^n$. Therefore, the last value must be greater than or equal to 1, hence:

$$\frac{b_0}{2^n} \geq 1$$

$$b_0 \geq 2^n \implies n \leq \log_2 b_0$$

Thus, if we take this in the form of time complexity as Big-O notation, we will get $n = O(\log b_0)$ and since a_0 is the larger number, we take $n = O(\log a)$.

Hence proved

This could have been proved considering the fibonacci series scenario as well as we will do in the B part.

Answer B

Worst case scenario for the Euclidean algorithm is when the two integers are part of the Fibonacci series. In this case, the algorithm takes the maximum number of steps. Let's consider F_n and F_{n-1} as the two numbers of the Fibonacci series. In the Euclidean algorithm, each step would reduce to the previous set of numbers in the series as we proceed by finding the mod:

$$(F_{n+1}, F_n) \to (F_n, F_{n-1}) \to (F_{n-1}, F_{n-2}) \to \ldots \to (F_2, F_1).$$

This will continue until we reach $(F_1, 0)$. As per the Fibonacci series growth rate, it grows exponentially as:

$$F_n \approx \frac{\phi^n}{\sqrt{5}}.$$

Therefore, if we take (F_{n+1}, F_n) in the form of the Euclidean theorem, i.e., $(F_{n+1}, F_n) = (a, b)$, then:

$$a = F_{n+1} \approx \frac{\phi^{n+1}}{\sqrt{5}}.$$

Taking the logarithm of both sides gives:

$$\log(a) \approx (n+1)\log\phi - \log(\sqrt{5}).$$

For large n, this simplifies to:

$$\log(a) \approx n \log\phi.$$

Thus, the number of steps n is proportional to $\log(a)$, specifically:

$$n \geq c\log a, \text{ for some constant } c > 0.$$

The Euclidean algorithm requires at least $c\log(a)$ steps for specific pairs of inputs, proving that the $O(\log a)$ bound is indeed tight.

4. TwoSum with Simple Restriction

- Given an array $A[0, \ldots, n-1]$ of integers, where $n \geq 3$.
- An integer target value T.

Objective: Design an algorithm to find indices i, j such that:

$$|A[i] + A[j] - T|$$

is minimized, with the following constraint:

- The indices i and j must satisfy $|i - j| > 1$.

Output:

- Return one pair of indices (i, j) that forms the sum closest to the target value T, subject to the constraint.

Requirements

1. **Algorithm:** Provide clear pseudocode that solves the problem.
2. **Correctness:** Prove the correctness of your algorithm by explaining why it works under the given constraint and guarantees the closest sum to the target.
3. **Running Time:** Analyze the running time of your algorithm and determine its time complexity.

Note: Full credit will be awarded for designing an algorithm with a time complexity of $O(n \log n)$.

Solution Explanation

To minimize $|A[i] + A[j] - T|$, we need to find the sum of two integers from the list which is closest to the target T, so that the absolute value can be minimized. We can create an array of indices for the given array. For example, let $A = [2, 8, 7, 6]$, and $T = 10$. The array of indices would be:

$$\text{Indices} = [0, 1, 2, 3]$$

When we sort the array A and the array of indices, we get:

$$A = [2, 6, 7, 8]$$

$$\text{Indices} = [0, 3, 2, 1]$$

We check the constraint that must be followed: $|i - j| > 1$.

As we iterate, there might be a case when we increase pointer i or decrease pointer j, we might miss a potential pair of indices that would have been the optimal answer.

To account for this, we iterate through the array A from both sides, keeping the pointer i from the start and pointer j from the last element of the array. If the constraint fails, we will store two values each in the direction of i and j, since an element can have at most 2 neighbors.

If $A[i] + A[j] < T$, we will increase the value of i; if $A[i] + A[j] > T$, we will decrease the value of j.

Pseudocode

```
function Minimized_Two_Sum(A: Array, T: Target):

    n <-- length(A)
    Index_A <-- [0, 1, ..., n-1] // Create array of indices
    Sort(A) <-- A
    Sort(Index_A) <-- Index_A  // Based on A
    i <-- 0 // Pointer i at the start
    j <-- n-1 // Pointer j at the last element
    min_diff <-- {infinity}
    temp_diff <-- {infinity}

    while i < j:

        // Checking the constraint with original indices
        if |Index_A[i] - Index_A[j]| > 1:
            if |A[i] + A[j] - T| < min_diff:
                min_diff <-- |A[i] + A[j] - T|
                x, y = Index_A[i], Index_A[j]


        // Check two values in the direction of i and j respectively
        else:
            if |A[i+1] + A[j] - T| < temp_diff and i+1 < j:
                temp_diff <-- |A[i+1] + A[j] - T|
                temp_x, temp_y <-- Index_A[i+1], Index_A[j]

            if |A[i+2] + A[j] - T| < temp_diff and i+2 < j:
                temp_diff <-- |A[i+2] + A[j] - T|
                temp_x, temp_y <-- Index_A[i+2], Index_A[j]

            if |A[i] + A[j-1] - T| < temp_diff and i < j-1:
                temp_diff <-- |A[i] + A[j-1] - T|
                temp_x, temp_y <-- Index_A[i], Index_A[j-1]

            if |A[i] + A[j-2] - T| < temp_diff and i < j-2:
                temp_diff <-- |A[i] + A[j-2] - T|
                temp_x, temp_y <-- Index_A[i], Index_A[j-2]

        if A[i] + A[j] < T:
            i = i + 1
        else:
            j = j - 1

    if temp_diff < min_diff:
        x, y = temp_x, temp_y

    return (x, y)
```

Time Complexity

Creating the new array for indices involves iterating through the array, which takes $O(n)$ time.

Sorting using quick or merge sort takes $O(n \log n)$ time.

Iterating through the sorted array takes $O(n)$ time since it includes only one loop and movement of the pointers.

Checking the constraints takes $O(1)$ time.

Overall complexity:

$$O(n) + O(n \log n) + O(n) + O(1) = O(n \log n)$$

Hence, the algorithm takes $O(n \log n)$ time.