|  |  |
| --- | --- |
| **Total Marks:** | **04** |
| **Obtained Marks:** |  |

**Design & Analysis of Algorithm**

**Assignment # 01**

**Submitted to: Dr. Shahzad Latif**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Submitted by: Ubaib Bin Waris**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Reg Number: 2212416**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ *Instructions****: Copied or shown assignments will be marked zero. Late submissions are not entertained in any case. Submit Hardcopy in classroom and softcopy on Google classroom.*

**Q1: Define pre-condition and post-condition in the context of algorithm correctness. Why are they important for ensuring that an algorithm works as intended?**

In the context of algorithm correctness, **pre-condition** and **post-condition** help make sure that an algorithm or piece of code works correctly by checking its inputs and outputs.

**Pre-condition**

A **pre-condition** is something that must be true **before** the algorithm runs. It’s like a requirement that needs to be met for the code to work properly. If this requirement isn’t met, the algorithm might not function correctly.

**Post-condition**

A **post-condition** is something that must be true **after** the algorithm finishes. It’s a check to make sure that the result or the state of the system is correct once the code is done.

**Why are they important?**

1. **Ensures the code works**: Pre-conditions and post-conditions make sure the code behaves as expected. If the pre-condition is satisfied before the code starts, the post-condition should be true after it finishes, showing that the code did its job correctly.
2. **Prevents errors**: Pre-conditions help avoid issues like dividing by zero or invalid inputs, ensuring the algorithm runs with the right conditions.
3. **Clear expectations**: These conditions provide a clear idea of what the code expects and what it will produce, making it easier to understand, test, and fix if something goes wrong.

**Q2: Explain the difference between partial correctness and total correctness of an algorithm. Provide an example to illustrate both concepts.**

The concepts of **partial correctness** and **total correctness** describe how well an algorithm performs its task, focusing on whether it gives the correct result and whether it completes its execution.

**Partial Correctness**

**Partial correctness** means that the algorithm will give the right answer if it finishes, but it doesn’t promise that it will always finish. The algorithm could work forever without stopping, but whenever it does stop, the result will be correct.

* **Key idea**: The algorithm gets the right answer, but it might not always stop.

**Total Correctness**

**Total correctness** means that the algorithm will always finish and give the right answer. This means two things: the algorithm won’t run forever, and when it stops, the result will be correct.

* **Key idea**: The algorithm always stops and always gives the right answer.

**Example**

Consider finding the greatest common divisor (GCD) of two numbers using the Euclidean algorithm.

1. **Partial correctness**: If the algorithm finishes, the GCD it gives will be correct. But if it gets stuck and never finishes (because of a coding mistake), it’s not fully reliable.
2. **Total correctness**: The algorithm will always stop after a few steps, and it will give the correct GCD for any two numbers.

**Partial correctness**: The answer will be right, but the algorithm might not finish.

**Total correctness**: The algorithm will always finish, and the answer will always be right.

**Q3: What is a loop invariant? Explain how it can be used to prove the correctness of an iterative algorithm with a simple example.**

A **loop invariant** is a rule or condition that stays true every time the loop runs in an algorithm. It helps show that the loop is doing what it’s supposed to, and it is useful for proving that an algorithm works correctly.

**How it works:**

To use a loop invariant to prove that an algorithm is correct, we check three things:

1. **Initialization**: Make sure the loop invariant is true before the loop starts.
2. **Maintenance**: Check that if the loop invariant is true at the start of a loop cycle, it remains true at the end of that cycle.
3. **Termination**: Show that when the loop ends, the loop invariant and the stopping condition together give the correct result.

**Example: Finding the sum of the first n numbers**

Let’s look at an example where we calculate the sum of the first n positive numbers, and we’ll use a loop invariant to prove that it works correctly.

**Algorithm**:

sum = 0

for i = 1 to n:

sum = sum + i

**Loop invariant**: After each loop, sum will always equal the sum of the numbers from 1 to i.

**Proving Correctness with a Loop Invariant:**

1. **Initialization**: Before the loop starts, sum is 0, which is correct because we haven’t added any numbers yet. So, the invariant is true.
2. **Maintenance**: During each loop, we add the current number i to sum. If sum is correct up to i-1, adding i makes sum correct up to i. So, the invariant stays true after each loop.
3. **Termination**: When the loop finishes, i equals n+1, meaning the loop has gone through all the numbers from 1 to n. The invariant tells us that sum is now the correct total of the first n numbers.

**Q4: Outline the general approach to proving the correctness of an iterative algorithm. Why is establishing the loop invariant important in this process?**

Proving the correctness of an iterative algorithm involves showing that it behaves as expected and produces the correct result. The general steps are:

1. **Understand the Goal**:
   * Clearly define what the algorithm is supposed to do. Know the input it receives and the output it should give.
2. **Find the Loop Invariant**:
   * Identify a condition that stays true before and after each loop cycle. This helps confirm that the algorithm is doing the right thing at every step.
3. **Check the Start (Initialization)**:
   * Show that the loop invariant is true before the loop begins (right at the start). This proves the algorithm starts off correctly.
4. **Check Every Step (Maintenance)**:
   * Prove that if the loop invariant is true at the start of a loop cycle, it will still be true at the end of that cycle. This ensures that the algorithm keeps doing the right thing throughout.
5. **Check When the Loop Ends (Termination)**:
   * Show that the loop eventually stops and that, when it does, the result is correct based on the loop invariant and the stopping condition.
6. **Conclusion**:
   * Once you've checked that the invariant holds throughout the loop and that the algorithm stops with the correct result, you can say the algorithm is correct.

**Why the Loop Invariant is Important**

Establishing a **loop invariant** is key because:

* **Shows Progress**: The loop invariant confirms that the algorithm is doing the right work step by step.
* **Keeps the Algorithm on Track**: It ensures that each part of the loop keeps things in order and that the algorithm isn't going off course.
* **Helps Prove Final Result**: When the loop stops, the loop invariant helps confirm that the final answer is correct.
* **Guarantees Correctness**: Without the loop invariant, it’s difficult to prove that the algorithm will give the right answer every time. It provides a clear structure for proving that the algorithm works at every step.

**Q5: Discuss the key assumptions made when analyzing the time complexity of algorithms in the RAM model. How do these assumptions impact the practical analysis of algorithms?**

When analyzing the time complexity of algorithms using the **RAM model** (Random Access Machine), several key assumptions are made. These assumptions simplify the analysis but may not always reflect how algorithms perform in the real world. Here’s a breakdown of the main assumptions and their impact on practical analysis.

**Key Assumptions in the RAM Model**

1. **Basic Operations Take Constant Time**:
   * The RAM model assumes that simple tasks like addition, subtraction, or comparing two numbers take the same amount of time, no matter how big the numbers are.

**Impact**: In reality, some operations can take longer, especially with very large numbers or more complex calculations. This can make the actual performance slower than expected.

1. **Unlimited Memory**:
   * It assumes there’s no limit to how much memory you can use, and that accessing any part of memory takes the same amount of time.

**Impact**: In practice, computers have limited memory, and reaching different memory locations can take varying amounts of time. For example, accessing data in the cache is usually faster than accessing it from main memory.

1. **Instructions Run Sequentially**:
   * The model assumes that instructions are executed one after the other, without any overlap or delays.

**Impact**: Modern computers often execute multiple instructions at the same time (thanks to features like multi-core processors), which can speed up execution. The RAM model doesn't account for this parallelism.

1. **Data Types Are Treated Equally**:
   * It treats all types of data (like integers and floats) the same, assuming they all take the same amount of time to process.

**Impact**: In real life, processing larger data types (like long numbers or strings) may take more time. This can affect how efficient an algorithm is when dealing with different types of data.

1. **Memory Management Is Free**:
   * The model assumes that allocating and freeing up memory takes no time or a constant amount of time.

**Impact**: In reality, managing memory (like using malloc in C or garbage collection) can introduce delays, affecting overall performance.

1. **No Memory Hierarchy**:
   * The RAM model assumes that there’s no difference between different types of memory (like cache, RAM, or disk storage).

**Impact**: Accessing data from different memory types can take very different amounts of time. For example, accessing data from the cache is much faster than from the hard disk.

**Impact of These Assumptions on Practical Analysis**

* **Simplified Understanding**: These assumptions help us focus on how algorithms behave in general, without getting into complex hardware details. However, this can lead to a gap between theoretical predictions and actual performance.
* **Real-World Variability**: An algorithm that seems efficient in theory might perform poorly on real machines due to factors like memory access speed or the time needed for complex operations. Conversely, some algorithms may perform better than expected because of hardware optimizations.
* **Ignoring Real Costs**: Since the RAM model assumes all operations are equal, it can underestimate the complexity of algorithms that rely heavily on slower operations, such as disk access or network communication.
* **Not Considering Modern Hardware**: The RAM model generally assumes a single-core processor. Since modern computers often use multi-core processors that can handle multiple tasks at once, this can make some algorithms much faster than the model suggests.