
# Week-20

Syed Muhammad haider razvi
M00832169

## CST2572

## Task 1) Timing code does not always show its true complexity as such but if you wanted to time code what features could you use?

While depending just on timing code does not always provide an accurate picture of its true difficulty due to issues such as hardware variances and compiler optimisations, the following features can be used to evaluate temporal complexity.

### 1. Code structure:

**Loops:** Nested loops have a larger temporal complexity than single loops. The number of iterations and operations done within each iteration have an impact on complexity.

**Conditional statements:** Complex conditional statements with several branches are more complex than simple ones**.**

**Function calls:** The complexity of the function being called, as well as the frequency of calls, all contribute to overall complexity**.**

### 2. Data Structures:

**Arrays:** Accessing and manipulating elements in an array normally takes constant time (O(1)), whereas looking for specific elements may need loops (O(n) in the worst case).

**Linked lists:** Operations like as insertion and deletion require traversing the list, which might result in linear time complexity.

**Hash tables** are efficient for search operations because accessing entries with a hash function takes a constant amount of time (O(1) on average).

### 3. Math Operations:

Basic arithmetic operations such as addition, subtraction, multiplication, and division take constant time (O(1)).

Logarithmic operations, such as log(n), have a logarithmic temporal complexity.

Exponential operations need exponential time complexity (O(2^n)).

### 4. Big O notation:

This often-used notation expresses the upper bound of an algorithm's temporal complexity in terms of input size. It allows for an asymptotic analysis of the algorithm's efficiency while neglecting constant components.

### 5. Profiling tools:

These tools can be used to determine the real execution time of various components of your code. They can assist detect performance bottlenecks and direct optimisation efforts. However, it is crucial to note that profiling findings might be influenced by variables other than the code's intrinsic complexity.

## Task 2) Why is the measure of complexity important to cryptography?

In cryptography, the measure of complexity is critical in establishing the security and feasibility of encryption methods. Here's why it matters:

### 1. Security:

**Resistance to brute-force attacks:** Complex algorithms take significantly longer for attackers to break merely by trying all conceivable keys or combinations. This increases the time and resources needed to break the encryption, rendering it computationally infeasible for most adversaries.

**Provable security:** Many current cryptographic algorithms rely on complexity-based security. This means that its security can be theoretically demonstrated using the premise that certain computational problems are difficult to solve. For example, public-key cryptography is based on the difficulty of factoring huge integers and solving the discrete logarithm problem.

### 2. Practicality:

**Efficiency of encryption and decryption:** While complicated algorithms provide greater security, they can be computationally expensive to implement. Balancing security and efficiency is critical, particularly in resource-constrained contexts.

Choosing algorithms with proper complexity levels enables efficient encryption and decryption while ensuring adequate security.

**Scalability:** Cryptographic systems frequently need to accommodate growing data quantities. Lower-complexity algorithms are more efficient for huge datasets, making them more useful in real-world applications.

Here's an example.

Consider two encryption algorithms: Algorithm A, having exponential complexity, and Algorithm B, with linear complexity.

While Algorithm A may be extremely secure, its long processing time may make it unsuitable for regular use.

Despite being faster, Algorithm B may not be secure enough for highly sensitive data due to its reduced complexity.

## Task 3) Consider how you could implement a stats option in the menu of your final assessment which contains information on timings etc. on functions used.

### I. Adding Stats Option in JavaScript Application:

While I cannot access the specifics of your final assessment owing to confidentiality, I can provide a broad framework and considerations for incorporating a statistics option into your JavaScript application's menu:

### 1. Function timings:

**Profiling Tools:** Use the profiling tools offered by your JavaScript environment (e.g., chrome.devtools.evaluate, console.profile() in some browsers), or create bespoke timing methods with timestamps (e.g., performance.now()).

**Capture timings:** Record the execution times of important functions while the assessment is running. Save these timings in an appropriate data structure (e.g., an object, array, or stats class).

### 2. Menu integration:

**Menu System:** Incorporate the statistics feature into your current menu system, including an option to retrieve these metrics.

**Accessing Data:** When the stats option is selected, obtain previously collected data and display it in an understandable fashion.

### 3. Data Presentation:

**User-Friendliness:** Present the information in a format that is easy to interpret and analyse, such as a table, chart, or any other appropriate visualisation approach.

**Customizability:** Allow users to filter, sort, or export data according to their preferences.

## II. Code Example (Illustrative):

```
// Profiling and data collection (replace with actual profiling logic)

const functionTimings = {};

function profileFunction(fnName, fn) {

 const startTime = performance.now();

 fn();

 const endTime = performance.now();

 functionTimings[fnName] = endTime - startTime;

}


// Sample functions (replace with your actual functions)

function function1() {

 // Simulate some computation

 for (let i = 0; i < 1000; i++) {

  Math.random(); // Simulate a simple operation

 }

}
```

```
function function2() {
  // Simulate some other computation
  for (let j = 0; j < 2000; j++) {
    Math.random(); // Simulate a simple operation
  }
}

// Profile functions (replace with actual profiling calls)
profileFunction("function1", function1);
profileFunction("function2", function2);

// Update function to create stats table (replace with actual menu integration)
function updateStatsTable() {
  const table = document.getElementById("statsTable");
  table.innerHTML = ""; // Clear existing content

  const headerRow = document.createElement("tr");
  const nameHeader = document.createElement("th");
  nameHeader.textContent = "Function Name";
  headerRow.appendChild(nameHeader);
  const timeHeader = document.createElement("th");
  timeHeader.textContent = "Execution Time (ms)";
  headerRow.appendChild(timeHeader);
  table.appendChild(headerRow);
```

```
  for (const [name, time] of Object.entries(functionTimings)) {
    const dataRow = document.createElement("tr");
    const nameCell = document.createElement("td");
    nameCell.textContent = name;
    dataRow.appendChild(nameCell);
    const timeCell = document.createElement("td");
    timeCell.textContent = time.toFixed(2); // Format time to two decimal places
    dataRow.appendChild(timeCell);
    table.appendChild(dataRow);
  }
}

// Call update function to populate the table (replace with menu interaction)
updateStatsTable();
```

## III. Key Points:

**Adapt Functionality:** Replace the example profiling logic and table update function with your application's implementation.

Integrate statistics processing and presentation into your current menu system and HTML framework.

**Data Security:** Take data security precautions when keeping or transferring profiling data, especially in production situations.

**Performance Considerations:** Profiling might have an impact on application performance. Assess trade-offs and optimise data collection and display based on your requirements.