TUTORIAL SESSION 21:

Time complexity of various problems

Concept Building

Time complexity is a critical concept in computer science that quantifies the amount of time an algorithm takes to complete as a function of the length of the input. It provides a way to analyze the efficiency of algorithms and helps in comparing different algorithms for the same problem.

Key Concepts of Time Complexity

1. **Definition**: Time complexity is expressed as a function of the size of the input, typically denoted as n. It describes how the runtime of an algorithm increases as the input size increases.
2. **Big O Notation**: The most common way to express time complexity is through Big O notation, which provides an upper bound on the time complexity of an algorithm. It describes the worst-case scenario of an algorithm's growth rate.
    - **Examples**:
        - O(1): Constant time – the algorithm's runtime does not change with the input size.
        - O(n): Linear time – the runtime increases linearly with the input size.
        - $O(n^2)$: Quadratic time – the runtime increases quadratically with the input size.
        - $O(2^n)$: Exponential time – the runtime doubles with each additional input element.
3. **Polynomial vs. Exponential Time**:
    - **Polynomial Time**: An algorithm is said to run in polynomial time if its time complexity can be expressed as $O(n^k)$ for some constant kk. Polynomial time is generally considered efficient and feasible for computation.
    - **Exponential Time**: An algorithm is said to run in exponential time if its time complexity can be expressed as $O(2^n)$ or similar. Exponential time algorithms are often impractical for large inputs due to their rapid growth.
4. **Classes of Problems**:
    - **P (Polynomial Time)**: The class of problems that can be solved by a deterministic Turing machine in polynomial time. Examples include sorting algorithms and searching algorithms.
    - **NP (Nondeterministic Polynomial Time)**: The class of decision problems for which a proposed solution can be verified in polynomial time. Examples include the Subset Sum problem and the Traveling Salesman problem.
    - **NP-Complete**: A subset of NP problems that are as hard as the hardest problems in NP. If any NP-complete problem can be solved in polynomial time, then all NP problems can be solved in polynomial time. Examples include the 3-SAT problem and the Hamiltonian Cycle problem.

  - **NP-Hard**: Problems that are at least as hard as NP-complete problems but do not need to be in NP. They may not even be decision problems. An example is the Halting Problem.

5. **Reduction**: This is a technique used to show that one problem is at least as hard as another. If problem A can be transformed into problem B in polynomial time, and if B is known to be hard, then A is also hard.

Examples of Time Complexity

1. **Linear Search**:
   - **Algorithm**: Search for an element in an unsorted list.
   - **Time Complexity**: O(n)– in the worst case, you may have to check every element.
2. **Binary Search**:
   - **Algorithm**: Search for an element in a sorted list.
   - **Time Complexity**: O(log n)– with each step, the search space is halved.
3. **Bubble Sort**:
   - **Algorithm**: A simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.
   - **Time Complexity**: $O(n^2)$ – in the worst case, every element needs to be compared with every other element.
4. **Quick Sort**:
   - **Algorithm**: A divide-and-conquer algorithm that sorts by selecting a 'pivot' and partitioning the array into elements less than and greater than the pivot.
   - **Time Complexity**: Average case O(n * log n), worst case $O(n^2)$ (when the smallest or largest element is always chosen as the pivot).
5. **Traveling Salesman Problem (TSP)**:
   - **Algorithm**: Find the shortest possible route that visits each city exactly once and returns to the origin city.
   - **Time Complexity**: The brute-force solution is O(n!), which is exponential and impractical for large n.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Pre-Tutorial (To be completed by student before attending tutorial session)

1. Consider a non-deterministic Turing machine that decides a language L. If the time complexity of this NDTM is O(n^k) for some constant k. Is L considered to be in P or NP? Explain.

Solution:

The language L is in **NP** because an NDTM decides it in polynomial time $O(n^k)$. However, we can't confirm L is in **P** without a deterministic polynomial-time algorithm.

2. **If a polynomial-time algorithm exists for any NP-complete problem, what can be concluded?**

Solution:

If a polynomial-time algorithm exists for any NP-complete problem, then **P = NP**.

3. What is the time complexity of the brute-force solution to the Subset Sum Problem?

Solution:

The time complexity is **O(2^n)**.

IN-TUTORIAL (To be carried out in presence of faculty in classroom)

1. Give an example of a problem that is in NP but not known to be NP-complete?

Solution:

An example of a problem that is in **NP** but not known to be **NP-complete** is **Graph Isomorphism**.

2. Compare polynomial-time and exponential-time algorithms with examples.

Solution:

Type	Definition	Example Problem	Time Complexity
Polynomial-time	Solves in $O(n^k)$ for some constant k	Sorting (e.g., Merge Sort)	$O(n \log n)$
Exponential-time	Solves in $O(2^n)$ or similar large growth	Subset Sum (brute-force)	$O(2^n)$

3. What is the time complexity of Dijkstra's algorithm when implemented with a priority queue?

Solution:

The time complexity of Dijkstra's algorithm with a priority queue is **O((V + E) log V).**

4. What is the time complexity of the DFS algorithm for a graph represented as an adjacency list?

Solution:

The time complexity of DFS using an adjacency list is **O(V + E).**

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Post-Tutorial (To be carried out by student after attending tutorial session)

1. Determine the time complexity of the following recursive function:

```
function fib(n):
  if n <= 1:
    return n
  else:
    return fib(n-1) + fib(n-2)
```

Solution:

The time complexity of the recursive Fibonacci function is **O(2^n)**.

2. Determine the time complexity of the following function:

```
def example_function(n):
for i in range(n):
for j in range(n):
print(i, j)
```

Solution:

- The time complexity of the function is **O(n^2)**.

3. Discuss the exponential time complexity of the Traveling Salesman Problem (TSP) using the brute force approach.

Solution:

The Traveling Salesman Problem (TSP) has a brute force time complexity of **O(n!)**. This is due to evaluating all permutations of cities, leading to impractical computation times as n increases.

Viva – Questions

1. What is time complexity, and why is it important in algorithm analysis?

Solution:

Time complexity measures an algorithm's efficiency as input grows.

2. Explain the difference between worst-case, average-case, and best-case time complexity.

Solution:

Worst-case: maximum time, average-case: expected time, best-case: minimum time.

(For Evaluator's use only)

Comment of the Evaluator (if Any)	Evaluator's Observation
	Marks Secured: out of **50** Full Name of the Evaluator: Signature of the Evaluator Date of Evaluation: