

SPPU-BE-COMP-CONTENT – KSKA Git

Total No. of Questions : 4]

SEAT No. :

P8465

[Total No. of Pages : 2

Oct-22/BE/Insem-41

B.E. (Computer Engineering)

DESIGN AND ANALYSIS OF ALGORITHMS

(2019 Pattern) (Semester - VII) (410241)

Time : 1 Hour]

[Max. Marks : 30

Instructions to the candidates:

- 1) Answer the question of Q.1 or Q.2, Q.3 or Q.4.
- 2) Neat diagrams must be drawn whenever necessary.
- 3) Figures to the right indicate full marks.
- 4) Assume suitable data, if necessary.

Q1) a) Why correctness of the algorithm is important? Define loop invariant property and prove the correctness of finding summation of n numbers using loop invariant property. [8]

b) What is iterative algorithm? Explain iterative algorithm design issues using examples. [7]

OR

Q2) a) How to prove that an algorithm is correct? How to prove the correctness of an algorithm using counter example? Give suitable example. [7]

b) Write a short note on any 4 problem solving strategies. [8]

Q3) a) What is Best, Average and Worst case Analysis of Algorithms? Analyse the following algorithm Best, Average and Worst case [8]

```
void sort (int a, int n) {  
    int i, j;  
    for (i = 0; i < n; i++) {  
        j = i-1;  
        key = a[i];  
        while (j >= 0 && a[j] > key)  
        {  
            a[j+1] = a[j];  
            j = j-1;  
        }  
        a[j+1] = key;  
    }  
}
```

P.T.O.

SPPU-BE-COMP-CONTENT – KSKA Git

- b) • Explain P, NP, NP-Hard and NP-Complete problems with examples.
- Explain 3-SAT problem using an example. Why is SAT so important in theoretical computer science?

[7]

OR

Q4) a) What is NP-complete class problem? How would you prove vertex cover problem is NP-complete class problem? [8]

b) What is Best, Average and Worst case Analysis of Algorithms? Analyse the following algorithm Best, Average and Worst case [7]

```
int Linear-search(int a, int n, int item) {  
    int i;  
    for (i = 0; i < n; i++) {  
        if (a[i] == item) {  
            return a[i]  
        }  
    }  
    return -1  
}
```

[illegible]

-> a)

Introduction:

- Algorithm correctness ensures that the algorithm produces the desired output for all valid inputs and behaves as expected in every condition.
- It is a fundamental requirement in algorithm design to guarantee reliability, accuracy, and stability in real-life systems and applications.

- Correctness of an algorithm means the algorithm must return the correct output for all possible valid inputs under defined constraints.
- This concept ensures that the logic behind the algorithm is not just syntactically right but also functionally valid in practice.

- Without correctness, an algorithm may yield wrong results even if it compiles and runs without errors.
- Incorrect algorithms may lead to failures in applications, especially in critical areas like banking, healthcare, and aviation systems.
- Validating correctness helps developers identify logical errors early and build confidence in algorithm implementation.
- It prevents unexpected system behaviors, crashes, or data loss caused by incorrect logic or faulty control flow.
- Ensuring correctness is a key step before optimizing algorithms for performance or deploying them into production systems.

- A loop invariant is a condition that holds true before and after each iteration of a loop during the algorithm's execution.



- Final value of sum is $1 + 2 + \dots + n$, which is correct.

E										
M										



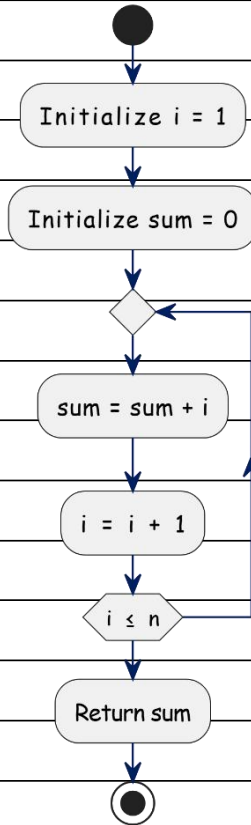
प्र.क्र./Q.No.

Q.1

diagram:

- figure : Loop Invariant for Sum of First n Numbers

figure : - Loop Invariant for Sum of First n Numbers

Explanation of diagram:

- The diagram describes the flow of summing numbers from 1 to n using a for loop with incrementing index i.
- The loop maintains the invariant that sum stores the total of all values from 1 to i-1 at the start of each cycle.
- After each addition and increment, the property still holds true, validating correctness by logical consistency.

Q.No.										TOTAL
E										
M										



प्र.क्र./Q.No.

Q.1

-> b) (Iterative Algorithm Design Issues)

Introduction:

- Iterative algorithms use repetition to solve problems by repeatedly executing a block of statements using loops like for, while, or do-while.
- These algorithms continue execution until a certain condition is met, making them suitable for problems with known steps or bounds.

Definition:

- Iterative algorithm:
 - An algorithm that repeats a specific block of instructions until a condition is satisfied is called an iterative algorithm.
 - Loop structures like for-loop, while-loop, or do-while-loop are used to implement such logic.
- Design issue:
 - Design issues refer to common challenges or concerns that must be addressed for correct and efficient algorithm behavior.

Key Characteristics of Iterative Algorithms:

- They are deterministic and repeat operations over a fixed or dynamically changing loop condition.
- Efficiency, termination, and correctness must be carefully handled during their design.

Common Design Issues in Iterative Algorithms:

- Loop Termination:
 - The algorithm must ensure the loop ends after a finite number of steps to prevent infinite loops.
 - If not handled, it can lead to a program crash or memory overflow.
- Correct Loop Condition:

[illegible]

Q.1

- ### Examples of Iterative Algorithms with Design Observations:

- *Example 1: Finding sum of first n natural numbers*

```
int sum = 0;  
for (int i = 1; i <= n; i++) {  
    sum = sum + i;  
}
```


[illegible]

Q.1

- }

- Iterative algorithms are generally easier to debug and understand than recursive ones.
- However, failure to manage loop conditions, initialization, and edge cases can result in design flaws.
- Choosing the correct loop construct (for, while) depends on the nature of the problem and known inputs.



(Proving Algorithm Correctness)

- Proving the correctness of an algorithm is essential to ensure that it works accurately under all possible valid inputs and conditions.
- It involves logically verifying that the algorithm produces the expected result by following well-defined steps or testing its failure through counter examples.

- Correctness of an algorithm means that for every input within the problem domain, the algorithm terminates and gives a correct output.
- Proof techniques such as mathematical induction, loop invariants, and counter examples are used to establish or question correctness.

- Proving correctness involves two key properties:
 - Partial correctness - the algorithm gives the correct result if it terminates.
 - Termination - the algorithm always finishes after a finite number of steps.
- Common methods used for correctness proof include:
 - Using loop invariants to track and prove correctness at each iteration.
 - Using mathematical induction to prove recursive or iterative behavior.
 - Counter examples to show specific cases where algorithm fails, thus proving it incorrect.

Definition: Counter Example in Algorithm Correctness:



Q.2

- Example: Incorrect Algorithm using Counter Example:

- ```
bool isEven(int x){
 return (x / 2) * 2 == x;
}
```

- Let's test it with a negative input  $x = -4$ .

Expected result: True, since -4 is even.

Output from algorithm:

- $(-4 / 2) = -2$ , and  $(-2 * 2) = -4$ , which gives  $-4 == -4 \rightarrow \text{True}$ .
- Now test with  $x = -3$ .

Expected result: False, since -3 is odd.

Output from algorithm:

- $(-3 / 2) = -1$  (integer division), and  $(-1 * 2) = -2$ , so  $-2 == -3 \rightarrow \text{False}$  (Correct output).
- Now modify the function as follows:

```
bool isEven(int x) {
 return x % 2 == 0;
}
```

- Use counter example for an incorrect version of algorithm like:

[illegible]

Q.2

}

- For  $x = 4$ ,  $(x \& 1) = 0 \rightarrow$  so result is False.  
But this logic wrongly detects even numbers as odd.

- The above algorithm fails for all even inputs and hence is incorrect.
- Just one counter example is enough to invalidate an algorithm's correctness completely.

## Introduction:

- In computer science, a *problem solving strategy* is a general method or technique used to design an algorithm that efficiently solves a given problem.
- Different strategies help to structure solutions and reduce the time and space complexity in various problem domains.

- *Problem solving strategy:*
  - It is a systematic approach used to analyze the problem and design an optimal or correct algorithmic solution.

- Divide and Conquer:
  - This strategy divides the problem into smaller subproblems, solves each recursively, and combines their solutions to solve the original problem.



Q.2

- diagram:

- figure : Divide and Conquer Strategy

```
graph TD; A["C Main Problem (Solve A)"] --> B["C Subproblem 1 (Solve A1)"]; A --> C["C Subproblem 2 (Solve A2)"]; A --> D["C Subproblem 3 (Solve A3)"]; B --> E["C Combine Results (A1 + A2 + A3)"]; C --> E; D --> E;
```

- The diagram shows how a large problem is divided into smaller ones, which are then solved and merged to get the final result.
- This method enhances performance in problems where smaller solutions are easier to compute.
- Dynamic Programming:
  - This method solves each subproblem once and stores its result to avoid redundant computations (overlapping subproblems).



- ### Additional Examples of Usage:

- Divide and Conquer in Binary Search:
  - Array is repeatedly divided in half until the desired value is found or all elements are checked.
- Greedy in Job Scheduling:
  - Select job with earliest finish time first to schedule maximum tasks.
- Dynamic Programming in Matrix Chain Multiplication:
  - Stores minimum cost for sub-chains and uses those to solve larger chain problems.
- Backtracking in N-Queens:
  - Places queens row by row, backtracks when two queens threaten each other.



-> a)

## Introduction:

- Algorithm analysis helps us predict the resources required (time or memory) as input size increases.
- Different scenarios arise depending on the input arrangement. These are called best case, worst case, and average case.

- *Best Case:*

- The minimum time an algorithm takes to run for any input. It occurs when the input is optimally arranged.
- Worst Case:
  - The maximum time an algorithm takes to run. It happens when the input is arranged in the least favorable order.
- Average Case:
  - The expected running time over all possible inputs. It gives a practical estimate of performance.

- Input size (n) refers to the total number of elements passed to the algorithm.
- The complexity is measured in terms of how many times loops or operations execute relative to n.

```
void sort (int a[], int n) {
 int i, j;
 for (i = 0; i < n; i++) {
```



}

- ### Summary of Complexities:





Q.3

- Best Case: Input is already sorted  $\rightarrow O(n)$
- Worst Case: Input is reverse sorted  $\rightarrow O(n^2)$
- Average Case: Input is random  $\rightarrow O(n^2)$

## Introduction:

- In complexity theory, problems are classified into different classes based on the resources required to solve them, especially time.
- The classes P, NP, NP-Hard, and NP-Complete help in understanding which problems can be solved or verified efficiently.

### Definition of P Class Problems:

- P class refers to all decision problems that can be solved in polynomial time using deterministic algorithms.
- These are the easiest problems that can be both solved and verified quickly.
- Example: Checking if a number is prime or not using the AKS algorithm belongs to class P.

### Definition of NP Class Problems:

- NP stands for Non-deterministic Polynomial time where solutions cannot be found quickly but can be verified in polynomial time.
- It is unknown whether NP problems can be solved in polynomial time like P problems.
- Example: The Subset Sum Problem where we check if a subset with a given sum exists.

### Definition of NP-Complete Problems:



- P: Linear Search, Binary Search, Matrix Multiplication
- NP: Travelling Salesman Decision Problem, Subset Sum
- NP-Complete: 3-SAT, Vertex Cover, Clique Problem
- NP-Hard: Halting Problem, General Optimization Versions of NP-Complete Problems





- SAT and 3-SAT problems are foundational in studying advanced complexity classes like NP, NP-complete, and NP-hard.
- They also serve as benchmarks for evaluating the efficiency of various heuristic and approximation algorithms.



## Introduction:

- In complexity theory, problems are classified into different classes based on the resources required to solve them, especially time.
- The classes P, NP, NP-Hard, and NP-Complete help in understanding which problems can be solved or verified efficiently.

- P class refers to all decision problems that can be solved in polynomial time using deterministic algorithms.
- These are the easiest problems that can be both solved and verified quickly.
- Example: Checking if a number is prime or not using the AKS algorithm belongs to class P.

- NP stands for Non-deterministic Polynomial time where solutions cannot be found quickly but can be verified in polynomial time.
- It is unknown whether NP problems can be solved in polynomial time like P problems.
- Example: The Subset Sum Problem where we check if a subset with a given sum exists.

- NP-Complete problems are a subset of NP problems that are as hard as any problem in NP.
- If any NP-Complete problem is solved in polynomial time, all NP problems can also be solved in polynomial time.
- These problems are both in NP and NP-Hard.



- *P*: Linear Search, Binary Search, Matrix Multiplication



- *NP*: Travelling Salesman Decision Problem, Subset Sum
- *NP-Complete*: 3-SAT, Vertex Cover, Clique Problem
- *NP-Hard*: Halting Problem, General Optimization Versions of NP-Complete Problems

### -> b) Best, Average and Worst Case Analysis of Linear Search Algorithm

## Introduction:

- The performance of an algorithm depends on the input it receives and may vary for different cases of the same input size.
- The three main types of performance analysis are: best case, average case, and worst case analysis.
- These cases help evaluate algorithm efficiency under various conditions and guide in selecting suitable algorithms.

*Definition:*

- Best Case: The minimum number of steps taken by an algorithm for any input size.
- Average Case: The expected number of steps considering all possible inputs, assuming they occur with equal probability.
- Worst Case: The maximum number of steps taken by an algorithm for any input size.

### Linear Search Algorithm Working:

- Linear search scans each element of the array one by one from index 0 to n-1.
- If the element is found, the function returns the element.
- If not found, it returns -1 indicating the item is not present in the array.

### Best Case Analysis:





Q.4

- ### Worst Case Analysis:

- ### Average Case Analysis:

- Diagram:

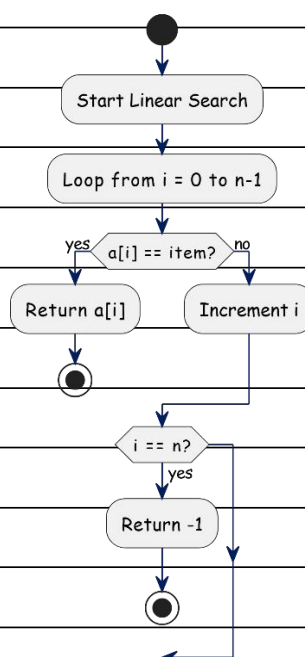


figure : Linear search control flow diagram



- The diagram shows a simple decision path where a loop runs through each element.
- If match is found early, it exits quickly (best case).
- If no match is found until end, it returns -1 (worst case).
- The same path is partially followed for average case depending on where the match is found.

- Best case performance of Linear Search is very fast with only one comparison.
- However, in practical scenarios where element may not be at the start, average or worst case dominates.
- Hence, for large datasets, more efficient search methods like binary search are preferred if data is sorted.