

# SPPU-BE-COMP-CONTENT – KSKA Git

## DAA Unit 1 – PYQ Answers

### ► October 2022

Q1)

a) Why correctness of the algorithm is important? [8 m]

- **Definition:**

The correctness of an algorithm means it produces the expected output for **all valid inputs** and stops after a finite number of steps.

- **Need of Correctness of Algorithm:**

1. To ensure that the algorithm is developed to correctly satisfy the functional requirements, there is a need for correctness of algorithm.
2. To ensure that the solution for the given problem is valid we need to check the correctness of algorithm.
3. To check the given problem has finite or infinite solution it is essential to check the correctness of algorithm.
4. The need for correctness of algorithm is for efficient execution of given task on computers.
5. To ensure that all the cases in the problem statements are covered, it is necessary to check the correctness of algorithm.
6. For some application correctness of algorithm means safety of human life or very costly equipment. Hence it is necessary to ensure that the algorithm is correct.

- **Basic steps in algorithmic correctness Following are the basic steps that need to be followed for checking the correctness of algorithm:**

1. Identification of the properties of input data. These properties of data are called preconditions.
2. Identification of the properties which must be satisfied by the output data. These properties are called postconditions.
3. Proving that starting from the preconditions and executing each step specified in the algorithms one obtains the postconditions.

- **Confirming correctness of algo:**

### Loop Invariant Property

- **Definition:**

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

- **Purpose:**

## SPPU-BE-COMP-CONTENT – KSKA Git

- Used to prove correctness of loops.
- Shows that the algorithm maintains a desired property until it terminates.

### Proving Correctness of Summation of n Numbers Using Loop Invariant Property \*

#### Algorithm:

Input: n (positive integer)

Output: Sum of first n natural numbers

sum  $\leftarrow$  0

for i  $\leftarrow$  1 to n do

    sum  $\leftarrow$  sum + i

end for

#### Loop Invariant Statement:

At the start of each iteration i of the loop, sum = 1 + 2 + ... + (i-1).

#### Proof:

1. **Initialization:** (Before first iteration, i = 1)
  - sum = 0, and there are no numbers before 1.
  - Property holds: sum = sum of first 0 numbers = 0.
2. **Maintenance:** (Assume property holds for iteration k)
  - Before iteration k+1, sum = 1 + 2 + ... + k.
  - In iteration k+1, we do: sum  $\leftarrow$  sum + (k+1).
  - Now sum = 1 + 2 + ... + k + (k+1), so property holds for iteration k+1.
3. **Termination:** (After last iteration, i = n+1)
  - sum = 1 + 2 + ... + n, which is the correct answer.

**Hence proved** that the algorithm correctly computes the sum of first n numbers.

**b) What is iterative algorithm? [7 m]**

**OR What is Iterative Algorithm? Explain Iterative Algorithm Design Issues with Examples.**

## SPPU-BE-COMP-CONTENT – KSKA Git

- **Definition:**

An **iterative algorithm** solves a problem by **repeatedly executing a set of instructions** using loops (for, while) until a condition is met.

- Iterative algorithm is an algorithm which has at least one iterative component or loop. Hence the part of algorithmic statements will be executed for n number of times.
- Even small amount of time spent on execution of loop will directly affect the efficiency of overall algorithm. This situation can be worst if nested loops (a loop within another loop) is present in the algorithm.
- **Example:** Linear search, finding factorial using loop, sum of n numbers.

### Algorithmic Design Issues for Iterative Algorithms

#### i) Correct Use of Loops in the Program

- Loops should be chosen according to the problem (e.g., for loop for fixed iterations, while loop for condition-based repetition).
- Ensure proper **initialization**, **condition checking**, and **update** steps to avoid infinite loops.
- Example:
- ```
for (int i = 0; i < n; i++) {  
    // loop body  
}
```

#### ii) Factors that Affect the Efficiency of Algorithm

- **Number of iterations** – More iterations increase time complexity.
- **Data size** – Larger input size generally increases execution time.
- **Operations inside the loop** – Complex operations inside loops slow execution.
- **Hardware & Compiler Optimization** – Processor speed and compiler optimizations can affect actual runtime.

#### iii) Estimation and Specification of Execution Time

- **Execution time** can be estimated by counting the total number of basic operations performed.

## SPPU-BE-COMP-CONTENT – KSKA Git

- Example:  
If a loop runs  $n$  times and each iteration performs a constant-time operation, execution time is proportional to  $n$ .
- Use **empirical testing** or theoretical analysis for estimating execution time.

### iv) Order Notations

- Used to express **time and space complexity** mathematically.
- **Big-O ( $O$ )** – Worst-case growth rate.
- **Omega ( $\Omega$ )** – Best-case growth rate.
- **Theta ( $\Theta$ )** – Average or exact growth rate.
- Example:  
If a loop runs  $n$  times, the time complexity is  **$O(n)$** .

### Example: Iterative Algorithm to Find Factorial

```
int factorial(int n) {  
    int fact = 1;  
    for (int i = 1; i <= n; i++) {  
        fact *= i;  
    }  
    return fact;  
}
```

- **Loop Type:** for loop
- **Time Complexity:**  $O(n)$
- **Space Complexity:**  $O(1)$

### Q2) a) How to prove that an algorithm is correct? [7 m]

- **Understand the Problem:** Know what the algorithm should do.
- **Specification:** Write preconditions (input conditions) and postconditions (expected output).
- **Proof of Correctness:**
  1. **Partial Correctness:** If algorithm stops, output is correct.

## SPPU-BE-COMP-CONTENT – KSKA Git

2. **Termination:** Algorithm finishes after finite steps.

- **Mathematical Proof Methods:**

- 1. **Loop Invariant Property**

- 2. **Mathematical Induction**

- 3. **Contradiction Method**

**Proving Correctness Example using Loop Invariant Property : < .....[Q1(a) above].....>**

**b) Short Note on Any 4 Problem Solving Strategies [8 m]**

- Problem solving strategy is a general approach by which many problems can be solved algorithmically. These problems may belong to different areas of computing. These strategies are also called as algorithmic techniques or algorithmic paradigms.

**1. Divide and Conquer:**

- Divide problem into subproblems, solve them recursively, combine results.
- Example: Merge Sort, Quick Sort.

**2. Greedy Method:**

- Make locally optimal choices hoping to find global optimum.
- Example: Kruskal's MST algorithm.

**3. Dynamic Programming:**

- Break problem into overlapping subproblems and store results to avoid recomputation.
- Example: Fibonacci using DP.

**4. Backtracking:**

- Explore all possibilities and backtrack when a solution is invalid.
- Example: N-Queens problem.

**5. Brute Force:**

- This is straightforward technique with naïve approach.
- Eg.: Bubble sort , Sequential sort

## SPPU-BE-COMP-CONTENT – KSKA Git

► September 2023

**Q1) a) Given the fastest computer and hypothetically infinite memory, do we still need to study algorithms? Justify. [2]**

**Yes, we still need to study algorithms, because:**

1. **Efficiency Matters:** Even with fast computers, inefficient algorithms (e.g.,  $O(n^2)$  when  $O(n \log n)$  is possible) will take excessive time for large inputs.
2. **Scalability:** Problems may grow beyond the capacity of even the fastest machines if the algorithm is inefficient.
3. **Optimal Resource Utilization:** Faster execution reduces energy consumption and operational costs.
4. **Real-world Constraints:** Practical systems have limits like network delay, disk I/O speed, and concurrency issues, which require efficient algorithms.

**b) How can we relate algorithms to technology? [6]**

**Relation:**

Algorithms form the **logical foundation** behind every technological process. They define the *step-by-step procedure* to achieve a task in a technology-driven system.

**Examples:**

1. **Search Engines (Google, Bing):** Use algorithms like PageRank to rank web pages.
2. **Data Compression:** ZIP files use Huffman coding and Lempel–Ziv algorithms.
3. **Navigation Apps (Google Maps):** Use Dijkstra's or A\* search for shortest path.
4. **Machine Learning:** Gradient Descent algorithm optimizes model training.
5. **Cybersecurity:** Encryption algorithms (AES, RSA) secure data.
6. **Social Media Feeds:** Recommendation algorithms personalize content.

**Conclusion:**

Technology is the *physical implementation*, while algorithms are the *logical blueprint*. Without algorithms, technology cannot function.

## SPPU-BE-COMP-CONTENT – KSKA Git

c) Consider an array A of n integers which are already in sorted order. Let x be the number being searched in the array A in a linear fashion. The code fragment performing this task is given below: [7]

*(Linear Search Code Analysis)*

**Given Code:**

```
int lin_search(int A[]) {  
    i = 0; flag = 0;  
    do {  
        if (x == A[i])  
            return (1); // Number found  
        else  
            i++;  
    } while (i < n);  
    return (0); // Number not found  
}
```

**i) Is this code efficient?**

- **Yes**, for **small input sizes** or when data is unsorted, linear search is a simple and efficient approach.
- However, given that the array is **already sorted**, **binary search** would be far more efficient ( $O(\log n)$  instead of  $O(n)$ ).
- Since the question says “We wish to use linear search only”, this is acceptable but **not optimal** for sorted data.

**ii) Design Issue with Iterative Algorithm:**

- **Termination Condition:** Loop stops when  $i == n$  or element found. Correctly avoids infinite loop.
- **No Early Exit Optimization for Sorted Array:**  
Since the array is sorted, the algorithm could terminate early if  $A[i] > x$ , but here it checks all elements unnecessarily if x is greater than all elements.
- **Efficiency Concern:** While fine for unsorted arrays, this design wastes operations for sorted data.

**Q2) a) ALREADY DONE**

## SPPU-BE-COMP-CONTENT – KSKA Git

### Q2) b) Correctness Proof of Square Algorithm Using Mathematical Induction [7]

#### Algorithm:

```
int sqr(int n) {  
    if (n == 0) return 0;  
    else return (2*n + sqr(n-1) - 1);  
}
```

#### To Prove:

For all  $n \geq 0$ ,  $\text{sqr}(n) = n^2$ .

#### Proof by Mathematical Induction:

##### Base Case: $n = 0$

- Algorithm returns 0, and  $0^2 = 0$ . Holds.

##### Induction Hypothesis:

Assume for some  $k \geq 0$ ,  $\text{sqr}(k) = k^2$  holds true.

##### Induction Step: Show $\text{sqr}(k+1) = (k+1)^2$

From algorithm:

$$\begin{aligned}\text{sqr}(k+1) &= 2*(k+1) + \text{sqr}(k) - 1 \\ &= 2k + 2 + \text{sqr}(k) - 1 \\ &= 2k + 1 + k^2 \quad [\text{By induction hypothesis: } \text{sqr}(k) = k^2] \\ &= k^2 + 2k + 1 \\ &= (k+1)^2\end{aligned}$$

Holds for  $k+1$ .

#### Conclusion:

By mathematical induction, the algorithm correctly computes  $n^2$  for all  $n \geq 0$ .



► **September 2024**

**Q1) a) Prove the correctness of Minimum Element Algorithm [7]**

**Algorithm:**

```
int min_element(int A[]) {  
    min = A[0];  
    for (i = 1; i < n; i++) {  
        if (A[i] < min)  
            min = A[i];  
    }  
    return min;  
}
```

**To Prove:** The algorithm returns the smallest element of array  $A[0..n-1]$ .

**Proof using Loop Invariant Property:**

**Loop Invariant Statement:**

At the start of each iteration  $i$  of the for loop, min contains the smallest value among  $A[0]$ ,  $A[1]$ , ...,  $A[i-1]$ .

**Steps:**

**1. Initialization:**

- Before the first iteration ( $i = 1$ ),  $\text{min} = A[0]$ .
- This is the smallest among  $A[0]$  (only one element so far).  
Property holds.

**2. Maintenance:**

- Assume the property holds for iteration  $i$ .
- At iteration  $i$ , compare  $A[i]$  with min.
  - If  $A[i] < \text{min}$ , update min to  $A[i]$ .
  - Else, keep min unchanged.
- In either case, min remains the smallest among  $A[0..i]$ .  
Property maintained.

**3. Termination:**

## SPPU-BE-COMP-CONTENT – KSKA Git

- After the last iteration ( $i = n$ ), min contains the smallest element among  $A[0..n-1]$ . Correctness proved.

### Time Complexity:

- Comparisons:  $n-1$
- Time Complexity:  $O(n)$

### Q2) a) Prove the correctness of Factorial Algorithm [7]

#### Algorithm:

```
int fact(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return (n * fact(n - 1));  
}
```

**To Prove:** For all  $n \geq 0$ ,  $\text{fact}(n)$  returns  $n!$ .

#### Proof using Mathematical Induction:

1. **Base Case:**  $n = 0$ 
    - Algorithm returns 1
    - By definition,  $0! = 1$
  2. **Induction Hypothesis:**  
Assume for some  $k \geq 0$ ,  $\text{fact}(k) = k!$  is correct.
  3. **Induction Step:**
  4.  $\text{fact}(k+1) = (k+1) * \text{fact}(k)$
  5.  $= (k+1) * k!$  [By hypothesis]
  6.  $= (k+1)!$
- Holds for  $k+1$ .

#### Conclusion:

By induction, the algorithm correctly computes  $n!$  for all  $n \geq 0$ .