# TYPES OF ALGORITHMS

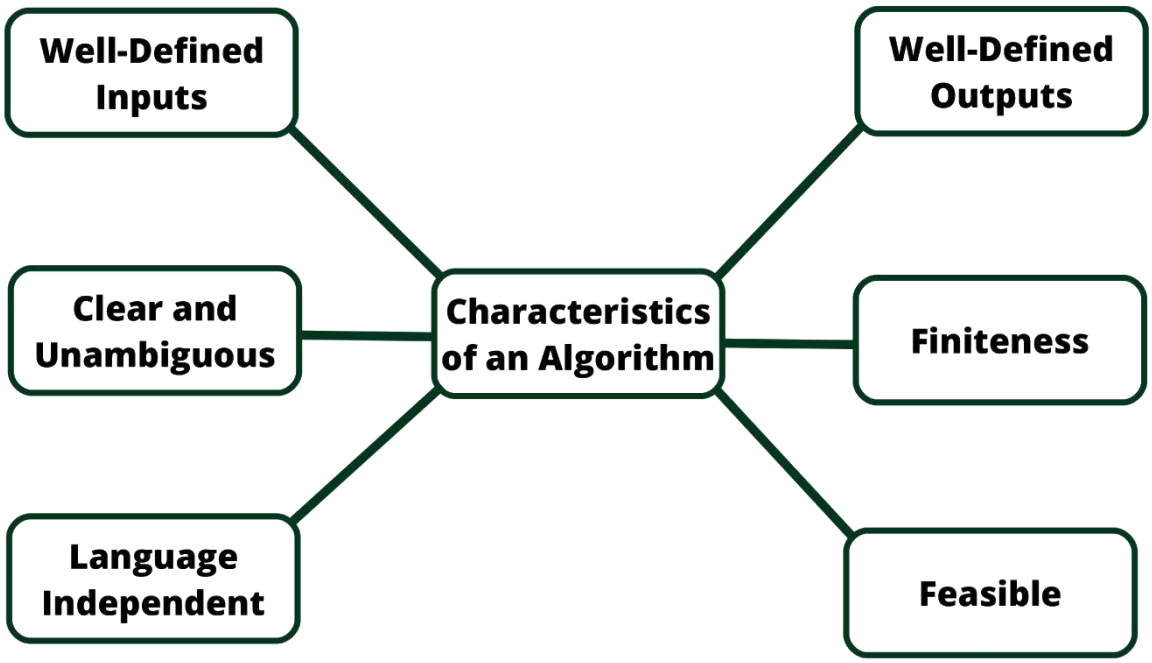
## Definition

An algorithm is a step-by-step procedure to solve a problem. It refers to the sequential steps and processes that should be followed to solve a problem

For example, you try cooking a new recipe, first you read the instructions and then follow the steps one by one as given in the recipe. Thus, after following the steps you will get your food ready.

Likewise, algorithms help to manage a task in programming to get the normal output. The algorithms designed are language-independent, that is they are just simple instructions that can be executed in any language. However, the output will be similar, as anticipated.

A good algorithm should be optimized in terms of time and space. Different types of problems require different types of algorithmic-techniques to be solved in the most optimized manner.




Let us look at a few examples of algorithms in our daily lives.

Take for instance, the task of making tea. Now the algorithm here would be the set of instructions one would follow so as to make this tea. So, the algorithm would include the following steps:

1. Heat water in a pan.
2. Add to it - even as the water is warming up - crushed ginger.
3. Add tea leaves.
4. Add milk.
5. When it comes to boil, add sugar.
6. Let it simmer for 2 to 3 minutes.

Following the above instructions would produce the desired result and solve our problem/fulfil the task.

The definition of algorithm in a formal context, however, alludes to its role in the way data is processed. In computer systems, an algorithm is an instance of logic written in software by software developers, and meant for the intended computer(s) to produce output from the given (sometimes zero) input.

So, to get a computer to do a task, we need to write a computer program. Doing so means telling the computer, step by step, exactly what we want it to do. The computer then runs/ reads the program and executes the commands shared in it by following each step mechanically so as to complete the said task. This series of related commands or steps is a computer algorithm.

It is important to note that algorithms are a finite sequence of well-defined, computer-implementable problem-solving instructions and that thus there is no room for ambiguity. Starting from an initial state and sometimes initial input, the algorithm spells out instructions that cover a finite number of well-defined successive states, eventually yielding output and terminating at a final ending state.

Now let us turn to a few examples of algorithms in basic math.

Take for instance, a problem where one has to figure out if a given number (say 7) is odd or even.

Here the algorithm would comprise the following steps:

1. Divide the number by 2. So here we divide 7 by 2.
2. Check for the remainder. If the remainder is 0, the number is even and if the remainder is not zero, the number is odd. Here the remainder would be 1. Hence the number 7 is an odd number.

Similarly, let us look at yet another example. Say the problem here is figuring out if the number 11 is a prime number. Now a prime number is one that is divisible only by 1 and itself.

So here the algorithm for this problem would have these steps.

1. Divide the number 11 by all numbers between 2 to 10 (since all numbers are divisible by 1 and themselves).
2. So first divide 11 by 2 and check for the remainder.
3. Then divide 11 by 3 and again check for remainder.
4. Continue to divide 11 by other numbers 4, 5 and so on till you divide 11 by the number 10. Check for the reminder after each division.
5. If the remainder after any of these divisions is 0, the number is not a prime number. If none of the remainders are 0, the number is a prime number. In this case, none of the remainders are zero. Hence, 11 is a prime number.

Just as in real life, in computer science and math as well, there are many types of algorithms available. In other words, often there are many ways - with different steps - of solving a given problem. All of these algorithms of course, need input to deliver a meaningful output.

Broadly speaking, algorithms, distinguished by their key features and functionalities, can be classified into seven categories:

- Brute Force algorithms
- Simple Recursive algorithms
- Backtracking algorithms
- Greedy algorithms
- Divide & Conquer algorithms
- Dynamic programming algorithms
- Randomised algorithms

# Explanation

## Brute force algorithms

A brute force algorithm involves blind iteration of all possible solutions to arrive at one or more solutions. Trial and error would be the method employed here; in other words, the solution lies in applying brute force, and hence the name of the algorithm.

Such an algorithm can be:

- Optimizing: Find the best solution. This may require finding all solutions, or if a value for the best solution is known, it may stop when any best solution is found
    - Example: Finding the best path for a travelling salesman
- Satisficing: Stop as soon as a solution is found that is good enough
    - Example: Finding a travelling salesman path that is within 10% of optimal

A simple example of a brute force algorithm would be trying to open a safe. Without any knowledge of the combination that can open the safe, the only way forward would be trying all possible combinations of numbers to open it. The same would be the case for someone trying to get access to another person's email account.

A brute force algorithm simply tries all possibilities until a satisfactory solution is found.

**Applications:**

- Linear Search
- Bubble Sort

**Example:**

```
public class LinearSearchExample{
public static int linearSearch(int[] arr, int key){
      for(int i=0;i<arr.length;i++){
         if(arr[i] == key){
            return i;
         }
      }
      return -1;
   }
   public static void main(String a[]){
      int[] a1= {10,20,30,50,70,90};
      int key = 50;
      System.out.println(key+" is found at index: "+linearSearch(a1, key));
   }
}
```

## Simple Recursive algorithms

A simple recursive algorithm:

- Solves the base cases directly
- Recurs with a simpler subproblem
- Does some extra work to convert the solution to the simpler subproblem into a solution to the given problem

Recursion simply means calling itself to solve its subproblems. These are called "simple" because several of the other algorithm types are inherently recursive.

Recursive algorithm is one which involves repetition of steps till the problem is solved. For instance, if one has to arrive at the greatest common factor (GCF) between any two numbers. Let's start with 14 and 18.

The algorithm here would be:

a) Divide 18 by 14.
b) Check the remainder (4).
c) Divide 14 by 4.
d) Check the remainder (2).
e) Divide 4 by 2.
f) Check the remainder (0).
g) Since the 0 remainder was arrived at upon division by the number 2, the GCF here would be 2.

Similarly, if one had to find the GCF between 12 and 16, one would go through similar steps but with the new inputs so as to arrive at the new result.

So, the algorithm here would be:

a) Divide 16 by 12.
b) Check the remainder (4).
c) Divide 12 by 4.
d) Check remainder (0).
e) Since the 0 remainder was arrived at upon division by the number 2, the GCF here would be 4.

As shown above, the process remains the same and is thus repeated, which makes this a recursive algorithm.

**Applications:**

- Recursive Insertion Sort
- Reverse a stack using recursion
- DFS
- Reversing a doubly linked list

**Example:**

```
public class FindGCDExample1
{
	public static void main(String[] args)
	{
		//x and y are the numbers to find the GCF
		int x = 12, y = 8, gcd = 1;
		//running loop form 1 to the smallest of both numbers
		for(int i = 1; i <= x && i <= y; i++)
		{
			//returns true if both conditions are satisfied
			if(x%i==0 && y%i==0)
			//storing the variable i in the variable gcd
			gcd = i;
		}
		//prints the gcd
		System.out.printf("GCD of %d and %d is: %d", x, y, gcd);
	}
}
```

## Backtracking algorithms

Backtracking algorithm is one that entails finding a solution in an incremental manner. There is often recursion/ repetition involved and attempts are made to solve the problem one part at a time. At any point, if one is unsuccessful at moving forward, one backtracks aka comes back to start over and find another way of reaching the solution. So backtracking algorithm solves a subproblem and if and when it fails to solve the problem, the last step is undone and one starts looking for the solution again from the previous point.

It is an improvement to the brute force approach. Here we start with one possible option out of many available and try to solve the problem if we are able to solve the problem with the selected move then we will print the solution else we will backtrack and select some other and try to solve it. It is a form of recursion, it's just that when a given option cannot give a solution, we

backtrack to the previous option which can give a solution, and proceed with other options.

An example would be when one plays chess. Typically, a good chess player contemplates the possible next move by the opponent in response to a certain move made by him/ her. Here each player is working out scenarios and often backtracking so as to arrive at the best possible way forward.

**Applications:**

- Hamiltonian Cycle
- N-Queens problem
- Rat in a maze problem

**Example:**

```
import java.util.*;
class GenerateBinaryStrings
{

// Function to print the output
static void printTheArray(int arr[], int n)
{
   for (int i = 0; i < n; i++)
   {
      System.out.print(arr[i]+" ");
   }
   System.out.println();
}

// Function to generate all binary strings
static void generateAllBinaryStrings(int n,
                     int arr[], int i)
{
   if (i == n)
   {
      printTheArray(arr, n);
      return;
   }
```

```
    // First assign "0" at ith position
    // and try for all other permutations
    // for remaining positions
    arr[i] = 0;
    generateAllBinaryStrings(n, arr, i + 1);

    // And then assign "1" at ith position
    // and try for all other permutations
    // for remaining positions
    arr[i] = 1;
    generateAllBinaryStrings(n, arr, i + 1);
}

// Driver Code
public static void main(String args[])
{
    int n = 4;

    int[] arr = new int[n];

    // Print all binary strings
    generateAllBinaryStrings(n, arr, 0);
}
}
```

## Greedy algorithms

In this algorithm, a decision is made that is good at that point without considering the future. This means that some local best is chosen and considers it as the global optimal. A greedy algorithm is a type of algorithm that is typically used for solving optimization problems. So whenever one wishes to extract the maximum in minimum time or with minimum resources, such an algorithm is employed.

There are two properties in this algorithm.

- Greedily choosing the best option
- Optimal substructure property: If an optimal solution can be found by retrieving the optimal solution to its subproblems.

Greedy Algorithm does not always work but when it does, it works like a charm! This algorithm is easy to devise and most of the time the simplest one.

Let us look at an example. Say person A is a reseller who has a bag that can carry a maximum weight of 20 pounds. Person A has been tasked with going to a warehouse and filling the bag to capacity in such a way as to maximise the profit upon selling those items. What items should A pick at the warehouse in order to maximise the eventual revenue/ profit? Here A would follow a series of steps (i.e., an algorithm) before arriving at a decision. A would likely do the following:

a) Look for the most expensive items that may also give him/ her a high markup
b) Check their size, volume and weight to evaluate how many such items can be accommodated in the bag.
c) Next look for the most in-demand items
d) Check their size, volume and weight to evaluate how many such items can be accommodated in the bag.
e) Consider all of the above then pick out the items.

In other words, person A would use the greedy algorithm here to get optimal solutions/ results. Here the optimal result would be picking out items that are not too big or heavy so as to be able to fit in the bag and at the same time, are fairly in demand and have a decent markup thus translating into higher revenues and profits.

**Applications:**

- Fractional Knapsack problem
- Prim's Algorithm
- Kruskal's Algorithm
- Huffman Coding

**Example:**

```
import java.util.Vector;
class FindMinimumCoins
{
   static int deno[] = {1, 2, 5, 10, 20,
   50, 100, 500, 1000};
   static int n = deno.length;
```

```
    static void findMin(int V)
    {
        // Initialize result
        Vector<Integer> ans = new Vector<>();

        // Traverse through all denomination
        for (int i = n - 1; i >= 0; i--)
        {
            // Find denominations
            while (V >= deno[i])
            {
                V -= deno[i];
                ans.add(deno[i]);
            }
        }

        // Print result
        for (int i = 0; i < ans.size(); i++)
        {
            System.out.print(
                " " + ans.elementAt(i));
        }
    }

    // Driver code
    public static void main(String[] args)
    {
        int n = 93;
        System.out.print(
            "Following is minimal number "
            +"of change for " + n + ": ");
        findMin(n);
    }
}
```

## Divide and Conquer algorithms

Another effective method of solving many problems, here, as the name suggests, one divides the steps, aka the algorithm into two parts. In the first part, the problem is broken into smaller subproblems of the same type. The second part is where the smaller problems are solved and then their solutions are considered together (combined) to produce the final solution of the problem.

A divide-and-conquer algorithm consists of two parts:

- Divide the problem into smaller subproblems of the same type, and solve these subproblems recursively
- Combine the solutions to the subproblems into a solution to the original problem

Traditionally, an algorithm is only called divide and conquer if it contains two or more recursive calls.

An example here would be a scenario where one has to find a student with a certain roll number - say 63 - in a school playground gathering. Now one way to do so would be to go about inquiring each child as to his/ her roll number. This, however, is not the quickest method.

A better way or algorithm would be one that goes like this:

a) Have the students line up in ascending or descending order of their respective roll numbers.
b) Split them into smaller groups - say of 50 students each - according to their roll numbers.
c) Find the student group with roll numbers 51 to 100.
d) Since 63 is less than the halfway (75) mark in this mark, go about inquiring in the first half of the group.
e) Continue to enquire till you reach student with roll number 63.

**Applications:**

- Binary Search
- Quicksort
- Mergesort

**Example:**

To find the maximum and minimum element in a given array.

```
class ArrayMaxMin {
        // Function to find the maximum no. in a given array.
        static int DAC_Max(int a[], int index, int l)
        {
           int max;

           if (index >= l - 2)
           {
              if (a[index] > a[index + 1])
                 return a[index];
              else
                 return a[index + 1];
           }

           // Logic to find the Maximum element in the given array.
           max = DAC_Max(a, index + 1, l);

           if (a[index] > max)
              return a[index];
           else
              return max;
        }

        // Function to find the minimum no. in a given array.
        static int DAC_Min(int a[], int index, int l)
        {
           int min;
           if (index >= l - 2)
           {
              if (a[index] < a[index + 1])
                 return a[index];
              else
                 return a[index + 1];
           }
```

```
        // Logic to find the Minimum element in the given array.
        min = DAC_Min(a, index + 1, l);

        if (a[index] < min)
            return a[index];
        else
            return min;
    }

    // Driver Code
    public static void main (String[] args)
    {

        // Defining the variables
        int min, max;

        // Initializing the array
        int a[] = { 70, 250, 50, 80, 140, 12, 14 };

        // Recursion - DAC_Max function called
        max = DAC_Max(a, 0, 7);

        // Recursion - DAC_Max function called
        min = DAC_Min(a, 0, 7);

        System.out.printf("The minimum number in " +
                          "a given array is: %d\n", min);
        System.out.printf("The maximum number in " +
                          "a given array is: %d", max);
    }
}
```

## Dynamic programming algorithms

A dynamic programming algorithm works by remembering the results of a previous run and using them to arrive at new results. Such an algorithm solves complex problems by breaking it into multiple simple subproblems, solving them one by one and storing them for future reference and use.

It simply means remembering the past and apply it to future corresponding results and hence this algorithm is quite efficient in terms of time complexity.

Dynamic Programming has two properties:

- **Optimal Substructure:** An optimal solution to a problem contains an optimal solution to its subproblems.
- **Overlapping subproblems**: A recursive solution contains a small number of distinct subproblems.

This algorithm has two versions:

- **Bottom-Up Approach:** Starts solving from the bottom of the problems i.e., solving the last possible subproblems first and using the result of those solving the above subproblems.
- **Top-Down Approach:** Starts solving the problems from the very beginning to arrive at the required subproblem and solve it using previously solved subproblems.

This differs from Divide and Conquer, where subproblems generally need not overlap.

A common example here would be finding a number in the Fibonacci series. The Fibonacci series has numbers that are the sum of the previous two numbers. So, if one was asked to share the fifth number in the Fibonacci series, one would arrive at the number 5 since the series would start as 1, 1, 2, 3 and then 5. Now if one were asked to calculate the seventh number in the series, the algorithm would typically have one build on the work done so far and take it forward. So in effect, one has remembered and used the results from the previous problem and deployed them to solve the current problem, thus arriving at the seventh number in the series, which is 13.

**Applications:**

- Fibonacci numbers
- Longest Common Subsequence
- 0-1 Knapsack problem

**Example:**

Ugly numbers are numbers whose only prime factors are 2, 3 or 5. The sequence 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, … shows the first 11 ugly numbers. By convention, 1 is included.

Given a number n, the task is to find $n^{th}$ Ugly number.

```
import java.lang.Math;

class UglyNumber
{
   // Function to get the nth ugly number
   int getNthUglyNo(int n)
   {
       // To store ugly numbers
      int ugly[] = new int[n];
      int i2 = 0, i3 = 0, i5 = 0;
      int next_multiple_of_2 = 2;
      int next_multiple_of_3 = 3;
      int next_multiple_of_5 = 5;
      int next_ugly_no = 1;

      ugly[0] = 1;

      for (int i = 1; i < n; i++)
      {
         next_ugly_no
            = Math.min(next_multiple_of_2,
                  Math.min(next_multiple_of_3,
                        next_multiple_of_5));

         ugly[i] = next_ugly_no;
         if (next_ugly_no == next_multiple_of_2)
         {
            i2 = i2 + 1;
            next_multiple_of_2 = ugly[i2] * 2;
         }
         if (next_ugly_no == next_multiple_of_3)
         {
            i3 = i3 + 1;
            next_multiple_of_3 = ugly[i3] * 3;
         }
         if (next_ugly_no == next_multiple_of_5)
         {
```

```
            i5 = i5 + 1;
            next_multiple_of_5 = ugly[i5] * 5;
          }
        }

        // End of for loop (i=1; i<n; i++)
        return next_ugly_no;
    }

    // Driver code
    public static void main(String args[])
    {

        int n = 150;

        // Function call
        UglyNumber obj = new UglyNumber();
        System.out.println(obj.getNthUglyNo(n));
    }
}
```

## Randomised algorithms

This is an algorithm type that makes its decision on the basis of random numbers i.e., it uses random numbers in its logic.

The best example of this is choosing the pivot element in quicksort. This randomness is to reduce time complexity or space complexity although not used regularly. Probability plays the most significant role in this algorithm.

In terms of quicksort, we fail to choose the correct element we might end up with a running time of O ($n^2$) in the worst case. Although if chosen with proper interpretation it can give the best running time of O (n log(n)).

**Applications**

- Randomised Quick Sort
- Kager's Algorithm etc.

# Problem Statements

## Linear search

**Problem:**

Given an array arr[] of n elements, write a function to search a given element x in arr[].

**Examples:**

Input: arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}

x = 110;

Output: 6

Element x is present at index 6

Input: arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}

x = 175;

Output: -1

Element x is not present in arr[].

## Factorial of a number

**Explanation:**

Factorial of a non-negative integer, is multiplication of all integers smaller than or equal to n. For example, factorial of 6 is 6*5*4*3*2*1 which is 720.

**Problem:**

Given a number n, find its factorial n!.

**Example:**

Input: 5

Output: Factorial of 5 is 120

## Binary search

**Explanation:**

The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n). Binary Search searches a sorted array by repeatedly dividing the search interval in half.

**Problem:**

Given a sorted array arr[] of n elements, write a function to search a given element x in arr[] using binary search.

**Examples:**

Input: arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}

x = 110;

Output: 6

Element x is present at index 6

Input: arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}

x = 175;

Output: -1

Element x is not present in arr[].

## Fibonacci numbers

**Explanation:**

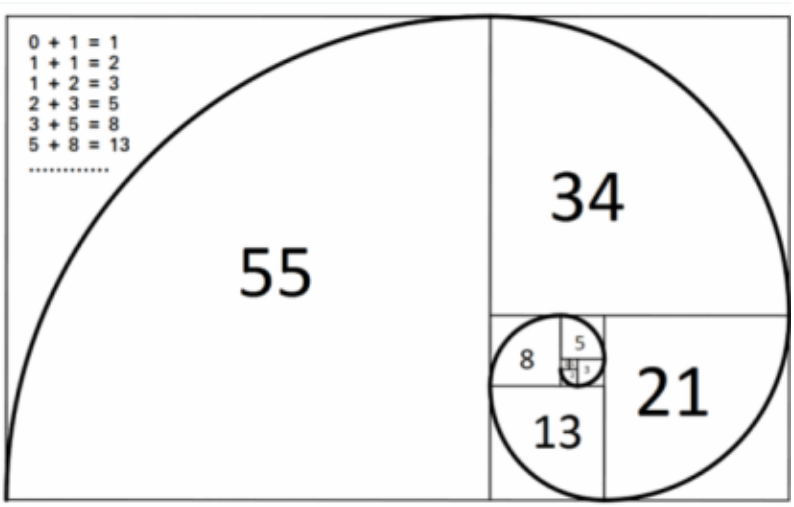
The Fibonacci numbers are the numbers in the following integer sequence.
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

In mathematical terms, the sequence $F_n$ of Fibonacci numbers is defined by the recurrence relation:

$$F_n = F_{n-1} + F_{n-2}$$

with seed values:

$$F_0 = 0 \text{ and } F_1 = 1.$$




**Problem:**

Given a number n, print $n^{th}$ Fibonacci Number.

Write a function *int fib (int n)* that returns $F_n$. For example, if *n* = 0, then *fib ()* should return 0. If n = 1, then it should return 1. For n > 1, it should return $F_{n-1} + F_{n-2}$

**Examples:**

Input: n = 2

Output: 1

Input: n = 9

Output: 34

## Assessment

1) An algorithm is _____
    a. A procedure for solving a problem
    b. A problem
    c. A real-life mathematical problem
    d. None of the mentioned

2) An algorithm in which we divide the problem into subproblem and then we combine the sub solutions to form solution to the original problem is known as _____
    a. Brute Force
    b. Divide and Conquer
    c. Greedy Algorithm
    d. None of the mentioned

3) An algorithm which uses the past results and uses them to find the new results is _____
    a. Brute Force
    b. Divide and Conquer
    c. Dynamic programming algorithms
    d. None of the mentioned

4) An algorithm which tries all the possibilities unless results are satisfactory is and generally is time-consuming is _____
    a. Brute Force
    b. Divide and Conquer
    c. Dynamic programming algorithms
    d. None of the mentioned

5) For a recursive algorithm _____
    a. a base case is necessary and is solved without recursion.
    b. a base case is not necessary
    c. a base case is necessary and is solved indirectly
    d. none of the mentioned

6) For an algorithm which is the most important characteristic that makes it acceptable _____

a. Fast
b. Compact
c. Correctness and Precision
d. None of the mentioned

7) There are two algorithms suppose A takes 1.41 milli seconds while B takes 0.9 milliseconds, which one of them is better considering all other things the same?
   a. A is better than B
   b. B is better than A
   c. Both are equally good
   d. None of the mentioned

8) An algorithm is a _____ set of precise instructions for performing computation.
   a. Infinite
   b. Finite
   c. Constant
   d. None of the mentioned

9) The Worst case occurs in linear search algorithm when _____
   a. Item is somewhere in the middle of the array
   b. Item is not in the array at all
   c. Item is the last element in the array
   d. Item is the last element in the array or is not there at all

10) Which of the following is a prerequisite for dynamic programming?
   a. Optimal substructure
   b. Overlapping subproblems
   c. Both a. and b.
   d. None of the above