**ASSIGNMENET-1**

1. **Explain Algorithm Specifications.**

An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

(1) Input. There are zero or more quantities that are externally supplied.

(2) Output. At least one quantity is produced.

(3) Definiteness. Each instruction is clear and unambiguous.

(4) Finiteness. If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.

(5) Effectiveness.

Every instruction must be basic enough to be carried out, in principle, by a person using only pencil and paper.

1. **Explain Time Complexity and Space complexity**

###### Time Complexity:

The time needed by an algorithm expressed as a function of the size of a problem is called the *time complexity* of the algorithm. The time complexity of a program is the amount of computer time it needs to run to completion.

The limiting behavior of the complexity as size increases is called the asymptotic time complexity. It is the asymptotic complexity of an algorithm, which ultimately determines the size of problems that can be solved by the algorithm.

###### Space Complexity:

The space complexity of a program is the amount of memory it needs to run to completion. The space need by a program has the following components:

**Instruction space:** Instruction space is the space needed to store the compiled version of the program instructions.

**Data space:** Data space is the space needed to store all constant and variable values. Data space has two components:

* + - Space needed by constants and simple variables in program.
    - Space needed by dynamically allocated objects such as arrays and class instances.

**Environment stack space:** The environment stack is used to save information needed to resume execution of partially completed functions.

**Instruction Space:** The amount of instructions space that is needed depends on factors such as:

* + - * The compiler used to complete the program into machine code.
      * The compiler options in effect at the time of compilation
      * The target computer.

1. **Explain 8 Queens Problem**

N-Queens Problem:

Let us consider, N = 8. Then 8-Queens Problem is to place eight queens on an 8 x 8 chessboard so that no two “attack”, that is, no two of them are on the same row, column, or diagonal.

All solutions to the 8-queens problem can be represented as 8-tuples (x1, . . . . , x8), where xi is the column of the ith row where the ith queen is placed.

The explicit constraints using this formulation are Si = {1, 2, 3, 4, 5, 6, 7, 8}, 1 < i <

1. Therefore the solution space consists of 88 8-tuples.

The implicit constraints for this problem are that no two xi‟s can be the same (i.e., all queens must be on different columns) and no two queens can be on the same diagonal.

This realization reduces the size of the solution space from 88 tuples to 8! Tuples.

The promising function must check whether two queens are in the same column or diagonal:

Suppose two queens are placed at positions (i, j) and (k, l) Then:

* + Column Conflicts: Two queens conflict if their xi values areidentical.
  + Diag 45 conflict: Two queens i and j are on the same 450 diagonal if:

i – j = k – l.

This implies, j – l = i – k

* + Diag 135 conflict:

i + j = k + l.

This implies, j – l = k – i

Therefore, two queens lie on the same diagonal if and only if:

j - l = i – k 

Where, j be the column of object in row i for the ith queen and l be the column of object in row „k‟ for the kth queen.

To check the diagonal clashes, let us take the following tile configuration:

**\***

**\***

**\***

**\***

**\***

**\***

**\***

**\***

In this example, we have:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| xi | 2 | 5 | 1 | 8 | 4 | 7 | 3 | 6 |

Let us consider for the case whether the queens on 3rd row and 8th row are conflicting or not. In this

case (i, j) = (3, 1) and (k, l) = (8, 6). Therefore:

j - l = i – k   1 - 6 = 3 – 8 

 5 = 5

In the above example we have, j - l = i – k , so the two queens are attacking. This is not a solution.

1. **Explain About Disjoint Set Operations**

Disjoint Set Operations Set:

A set is a collection of distinct elements. The Set can be represented, for examples, as S1={1,2,5,10}.

Disjoint Sets:

The disjoints sets are those do not have any common element.

For example S1={1,7,8,9} and S2={2,5,10}, then we can say that S1 and S2 are two disjoint sets.

Disjoint Set Operations:

The disjoint set operations are

* 1. Union
  2. Find

Disjoint set Union:

If Si and Sj are tow disjoint sets, then their union Si U Sj consists of all the elements x such that x is in Si or Sj.

Example:

S1={1,7,8,9} S2={2,5,10}

S1 U S2={1,2,5,7,8,9,10}

Disjoint Union:

To perform disjoint set union between two sets Si and Sj can take any one root and make it sub-tree of the other. Consider the above example sets S1 and S2 then the union of S1 and S2 can be represented as any one of the following.

Find:

To perform find operation, along with the tree structure we need to maintainthe name of each set. So, we require one more data structure to store the set names. The data structure contains two fields. One is the set name and the other one is the pointer to root.

**ASSIGNMENET-2**

1. **What are the applications of BackTracking**

Applications of Backtracking Algorithm

The backtracking algorithm has the following applications:

### 1. To Find All Hamiltonian Paths Present in a Graph.

A Hamiltonian path, also known as a Hamilton path, is a graph path connecting two graph vertices that visit each vertex exactly once. If a Hamiltonian way exists with adjacent endpoints, the resulting graph cycle is a Hamiltonian or Hamiltonian cycle.

**2.** To Solve the N Queen Problem**.**

* The problem of placing n queens on the nxn chessboard so that no two queens attack each other is known as the n-queens puzzle.
* Return all distinct solutions to the n-queens puzzle given an integer n. You are free to return the answer in any order.
* Each solution has a unique board configuration for the placement of the n-queens, where 'Q' and. '' represent a queen and a space, respectively.

### 3. Maze Solving Problems

There are numerous maze-solving algorithms, which are automated methods for solving mazes. The random mouse, wall follower, Pledge, and Trémaux's algorithms are used within the maze by a traveler who has no prior knowledge of the maze. In contrast, a person or computer programmer uses the dead-end filling and shortest path algorithms to see the entire maze at once.

### 4. The Knight's Tour Problem

The Knight's tour problem is the mathematical problem of determining a knight's tour. A common problem assigned to computer science students is to write a program to find a knight's tour. Variations of the Knight's tour problem involve chess boards of different sizes than the usual n x n irregular (non-rectangular) boards.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

1. **Explain Non Deterministic Algorithm**

A non-deterministic algorithm can provide different outputs for the same input on different executions. Unlike a deterministic algorithm which produces only a single output for the same input even on different runs, a non-deterministic algorithm travels in various routes to arrive at the different outcomes.

Non-deterministic algorithms are useful for finding approximate solutions, when an exact solution is difficult or expensive to derive using a deterministic algorithm.

One example of a non-deterministic algorithm is the execution of concurrent algorithms with race conditions, which can exhibit different outputs on different runs. Unlike a deterministic algorithm which travels a single path from input to output, a non-deterministic algorithm can take many paths, with some arriving at the same outputs, and others arriving at different outputs. This feature is mathematically used in non-deterministic computation models like non-deterministic finite automaton.

A non-deterministic algorithm is capable of execution on a deterministic computer which has an unlimited number of parallel processors. A non-deterministic algorithm usually has two phases and output steps. The first phase is the guessing phase, which makes use of arbitrary characters to run the problem.

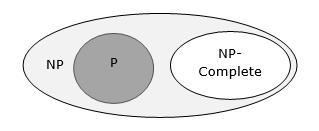
The second phase is the verifying phase, which returns true or false for the chosen string. There are many problems which can be conceptualized with help of non-deterministic algorithms including the unresolved problem of P vs NP in computing theory.

Non-deterministic algorithms are used in solving problems which allow multiple outcomes. Every outcome the non-deterministic algorithm produces is valid, regardless of the choices made by the algorithm during execution.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

1. **What are Np hard and Np complete**

A problem is in the class NPC if it is in NP and is as **hard** as any problem in NP. A problem is **NP-hard** if all problems in NP are polynomial time reducible to it, even though it may not be in NP itself.



If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable. These problems are called **NP-complete**. The phenomenon of NP-completeness is important for both theoretical and practical reasons.

## Definition of NP-Completeness

A language **B** is ***NP-complete*** if it satisfies two conditions

* **B** is in NP
* Every **A** in NP is polynomial time reducible to **B**.

If a language satisfies the second property, but not necessarily the first one, the language **B** is known as **NP-Hard**. Informally, a search problem **B** is **NP-Hard** if there exists some **NP-Complete** problem **A** that Turing reduces to **B**.

The problem in NP-Hard cannot be solved in polynomial time, until **P = NP**. If a problem is proved to be NPC, there is no need to waste time on trying to find an efficient algorithm for it. Instead, we can focus on design approximation algorithm

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. **Explain Applications of Dynamic Programming**

Dynamic programming is a technique that breaks the problems into sub-problems, and saves the result for future purposes so that we do not need to compute the result again. The subproblems are optimized to optimize the overall solution is known as optimal substructure property. The main use of dynamic programming is to solve optimization problems. Here, optimization problems mean that when we are trying to find out the minimum or the maximum solution of a problem. The dynamic programming guarantees to find the optimal solution of a problem if the solution exists.

**Applications of Dynamic Programming**

1. **OptimalBinary sSearch Tree**
2. **0/1 Knapsack problem**
3. **All Pairs Shortest Path**
4. **Reliabilty Design**
5. **Travelling Sales man Problem**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**