UNIT 4

Computable Problems – You are familiar with many problems (or functions) that are computable (or decidable), meaning there exists some algorithm that computes an answer (or output) to any instance of the problem (or for any input to the function) in a finite number of simple steps. A simple example is the integer increment operation:

$f(x) = x + 1$

It should be intuitive that given any integer x, we can compute x + 1 in a finite number of steps. Since x is finite, it may be represented by a finite string of digits. Using the addition method (or algorithm) we all learned in school, we can clearly compute another string of digits representing the integer equivalent to x + 1. Yet there are also problems and functions that are non-computable (or undecidable or uncomputable), meaning that there exists no algorithm that can compute an answer or output for all inputs in a finite number of simple steps. (Undecidable simply means non-computable in the context of a decision problem, whose answer (or output) is either "true" or "false").

Non-Computable Problems – A non-computable is a problem for which there is no algorithm that can be used to solve it. The most famous example of a non-computability (or undecidability) is the Halting Problem. Given a description of a Turing machine and its initial input, determine whether the program, when executed on this input, ever halts (completes). The alternative is that it runs forever without halting. The halting problem is about seeing if a machine will ever come to a halt when a certain input is given to it or if it will finish running. This input itself can be something that keeps calling itself forever, which means that it will cause the program to run forever. Another example of an uncomputable problem is: determining whether a computer program loops forever on some input. You can replace "computer program" with "Turing machine or algorithm" if you know about the Turing machine.

Proving Computability or Non-Computability – We can show that a problem is computable by describing a procedure and proving that the procedure always terminates and always produces the correct answer. It is enough to provide a convincing argument that such a procedure exists. Finding the actual procedure is not necessary (but often helps to make the argument more convincing). To show that a problem is not computable, we need to show that no algorithm exists that solves the problem. Since there are an infinite number of possible procedures, we cannot just list all possible procedures and show why each one does not solve the problem. Instead, we need to construct an argument showing that if there were such an

algorithm, it would lead to a contradiction. The core of our argument is based on knowing the Halting Problem is non-computable. If a solution to some new problem P could be used to solve the Halting Problem, then we know that P is also non-computable. That is, no algorithm exists that can solve P since if such an algorithm exists, it could be used to also solve the Halting Problem, which we already know is impossible. The proof technique where we show that a solution for some problem P can be used to solve a different problem Q is known as a reduction.A problem Q is reducible to a problem P if a solution to P could be used to solve Q. This means that problem Q is no harder than problem P since a solution to problem Q leads to a solution to problem P.

Some Examples Of Computable Problems – These are four simple examples of the computable problem:

Computing the greatest common divisor of a pair of integers.

Computing the least common multiple of a pair of integers.

Finding the shortest path between a pair of nodes in a finite graph.

Determining whether a propositional formula is a tautology.

Some Examples Of Computable Problems – State Entry Problem. Consider the problem of determining if a string 'w' is given to some Turing machine 'M' will it enter some state 'q'(where q belongs to a set of all states in Turing machine M and string w is not equal to an empty string). Is it computable or non-computable?

Explanation – We show the State Entry Problem is non-computable by showing that it is as hard as The Halting Problem, which we already know is non-computable. State Entry Problem is asking us on given string w if we start from initial state of Turing machine will it reach to a state q. Now this state entry problem can be converted to a halting problem. Halting problem is whether our Turing machine ever halts, and the state entry problem is asking the same thing whether this Turing machine halts at some state q if we give string w as input to the Turing machine M. So the state entry problem is non-computable as we converted it to the halting problem which we already know is non-computable problem. So in this way, we can prove non-computability.

Computable Problems:

Computable problems are those that can be solved by an algorithm, a step-by-step procedure that can be executed by a computer. These problems have well-defined inputs and outputs,

and there exists a deterministic algorithm that can produce the correct output for any given input.

Advantages

Efficiency: Computable problems can be solved efficiently using algorithms. There exist efficient algorithms for many computable problems, allowing for practical solutions.

Predictability: Since computable problems have well-defined inputs and outputs, their behavior is predictable. This makes it easier to reason about their properties and analyze their performance.

Disadvantages:

Limitations: Not all problems are computable. There are certain problems, such as the Halting Problem, which cannot be solved by any algorithm. These problems have undecidable properties, meaning there is no algorithm that can always produce the correct output.

Complexity: Although computable problems can be solved, the efficiency of the algorithms can vary significantly. Some problems may have solutions that are computationally expensive, requiring a large amount of time or resources to compute the result

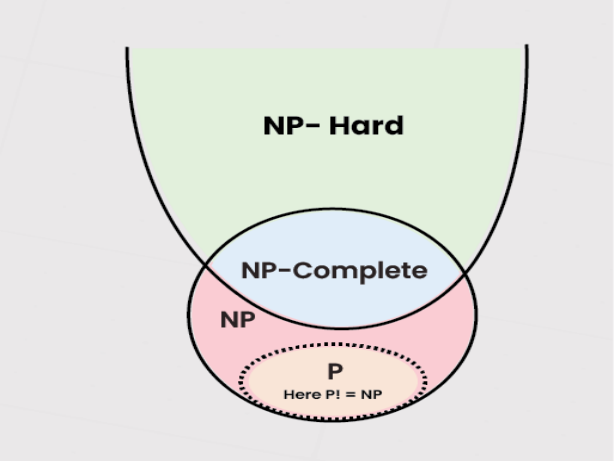
P, NP, CoNP, NP hard and NP complete | Complexity Classes

In computer science, there exist some problems whose solutions are not yet found, the problems are divided into classes known as **Complexity Classes**. In complexity theory, a Complexity Class is a set of problems with related complexity. These classes help scientists to group problems based on how much time and space they require to solve problems and verify the solutions. It is the branch of the theory of computation that deals with the resources required to solve a problem.

The common resources are time and space, meaning how much time the algorithm takes to solve a problem and the corresponding memory usage.

- The time complexity of an algorithm is used to describe the number of steps required to solve a problem, but it can also be used to describe how long it takes to verify the answer.
- The space complexity of an algorithm describes how much memory is required for the algorithm to operate.

Complexity classes are useful in organising similar types of problems.



Types of Complexity Classes

This article discusses the following complexity classes:

1. **P Class**
2. **NP Class**
3. **Co-NP Class**
4. **NP-hard**
5. **NP-complete**

P Class

The P in the P class stands for **Polynomial Time.** It is the collection of decision problems(problems with a "yes" or "no" answer) that can be solved by a deterministic machine in polynomial time.

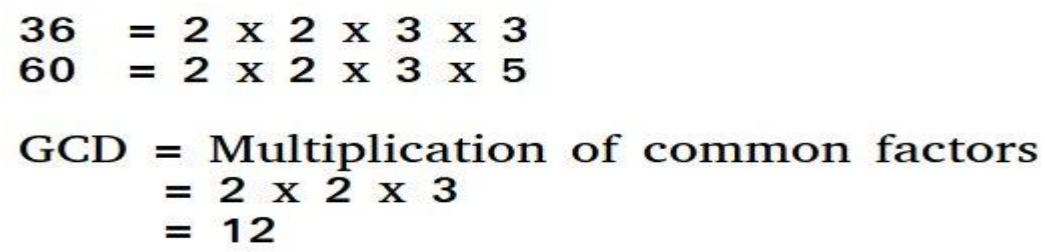
Features:

- The solution to **P problem**s is easy to find.
- **P** is often a class of computational problems that are solvable and tractable. Tractable means that the problems can be solved in theory as well as in practice. But the problems that can be solved in theory but not in practice are known as intractable.

This class contains many problems:

1. **Calculating the greatest common divisor.**

   The Euclidean algorithm is a way to find the greatest common divisor of two positive integers. GCD of two numbers is the largest number that divides both of them. A simple way to find GCD is to factorize both numbers and multiply common prime factors.



Basic Euclidean Algorithm for GCD:

The algorithm is based on the below facts.

- If we subtract a smaller number from a larger one (we reduce a larger number), GCD doesn't change. So if we keep subtracting repeatedly the larger of two, we end up with GCD.
- Now instead of subtraction, if we divide the larger number, the algorithm stops when we find the remainder 0.

Below is a recursive function to evaluate gcd using Euclid's algorithm:

```
// C++ program to demonstrate
// Basic Euclidean Algorithm

#include <bits/stdc++.h>
using namespace std;

// Function to return
// gcd of a and b
int gcd(int a, int b)
{
    if (a == 0)
        return b;
    return gcd(b % a, a);
}

// Driver Code
int main()
{
    int a = 10, b = 15;
```

```
    // Function call
    cout << "GCD(" << a << ", " << b << ") = " << gcd(a, b)
         << endl;
    a = 35, b = 10;
    cout << "GCD(" << a << ", " << b << ") = " << gcd(a, b)
         << endl;
    a = 31, b = 2;
    cout << "GCD(" << a << ", " << b << ") = " << gcd(a, b)
         << endl;
    return 0;
}
```

Output

GCD(10, 15) = 5

GCD(35, 10) = 5

GCD(31, 2) = 1

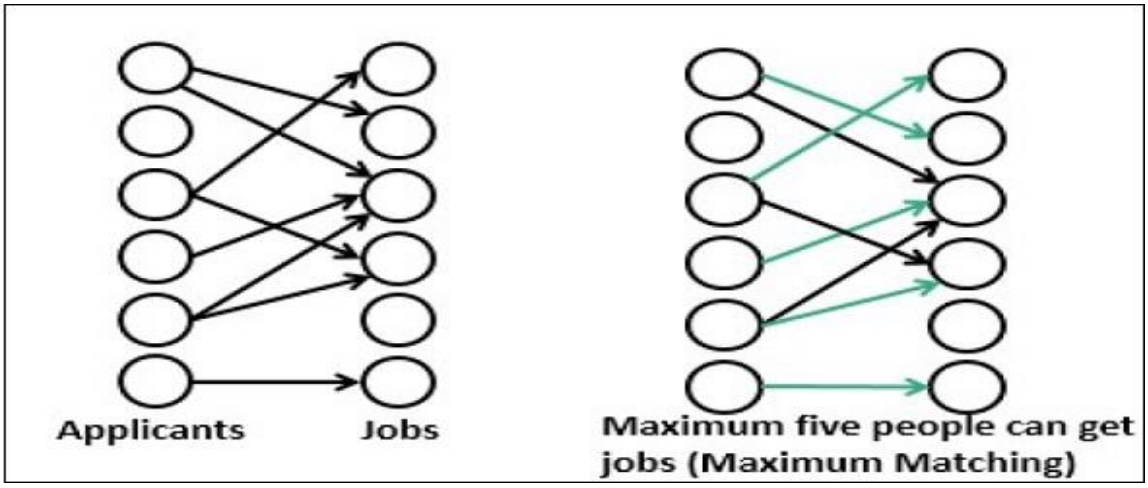
Time Complexity: O(Log min(a, b))

Auxiliary Space: O(Log (min(a,b))

2. **Finding a maximum matching.**

Maximum Bipartite Matching

The bipartite matching is a set of edges in a graph is chosen in such a way, that no two edges in that set will share an endpoint. The maximum matching is matching the maximum number of edges.



When the maximum match is found, we cannot add another edge. If one edge is added to the maximum matched graph, it is no longer a matching. For a bipartite graph, there can be more than one maximum matching is possible.

Input and Output

Input:

The adjacency matrix.

0 1 1 0 0 0

1 0 0 1 0 0

0 0 1 0 0 0

0 0 1 1 0 0

0 0 0 0 0 0

0 0 0 0 0 1

Output:

Maximum number of applicants matching for job: 5

Algorithm

bipartiteMatch(u, visited, assign)

Input: Starting node, visited list to keep track, assign the list to assign node with another node.

Output − Returns true when a matching for vertex u is possible.

```
Begin
   for all vertex v, which are adjacent with u, do
      if v is not visited, then
         mark v as visited
```

```
      if v is not assigned, or bipartiteMatch(assign[v], visited, assign) is true, then
        assign[v] := u
        return true
  done
  return false
End
```

maxMatch(graph)

Input − The given graph.

Output − The maximum number of the match.

```
Begin
  initially no vertex is assigned
  count := 0
  for all applicant u in M, do
    make all node as unvisited
    if bipartiteMatch(u, visited, assign), then
      increase count by 1
  done
End
```


Example

```
#include <iostream>
#define M 6
#define N 6
using namespace std;

bool bipartiteGraph[M][N] = {    //A graph with M applicant and N jobs
  {0, 1, 1, 0, 0, 0},
  {1, 0, 0, 1, 0, 0},
  {0, 0, 1, 0, 0, 0},
  {0, 0, 1, 1, 0, 0},
```

```
  {0, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 0, 1}
};

bool bipartiteMatch(int u, bool visited[], int assign[]) {
  for (int v = 0; v < N; v++) {    //for all jobs 0 to N-1
    if (bipartiteGraph[u][v] && !visited[v]) {    //when job v is not visited and u is interested
      visited[v] = true;    //mark as job v is visited
      //when v is not assigned or previously assigned
      if (assign[v] < 0 || bipartiteMatch(assign[v], visited, assign)) {
        assign[v] = u;    //assign job v to applicant u
        return true;
      }
    }
  }
  return false;
}

int maxMatch() {
  int assign[N];    //an array to track which job is assigned to which applicant
  for(int i = 0; i<N; i++)
    assign[i] = -1;    //initially set all jobs are available
  int jobCount = 0;

  for (int u = 0; u < M; u++) {    //for all applicants
    bool visited[N];
    for(int i = 0; i<N; i++)
      visited[i] = false;    //initially no jobs are visited
    if (bipartiteMatch(u, visited, assign))    //when u get a job
      jobCount++;
  }
  return jobCount;
}
```

```
int main() {
  cout << "Maximum number of applicants matching for job: " << maxMatch();
}
```

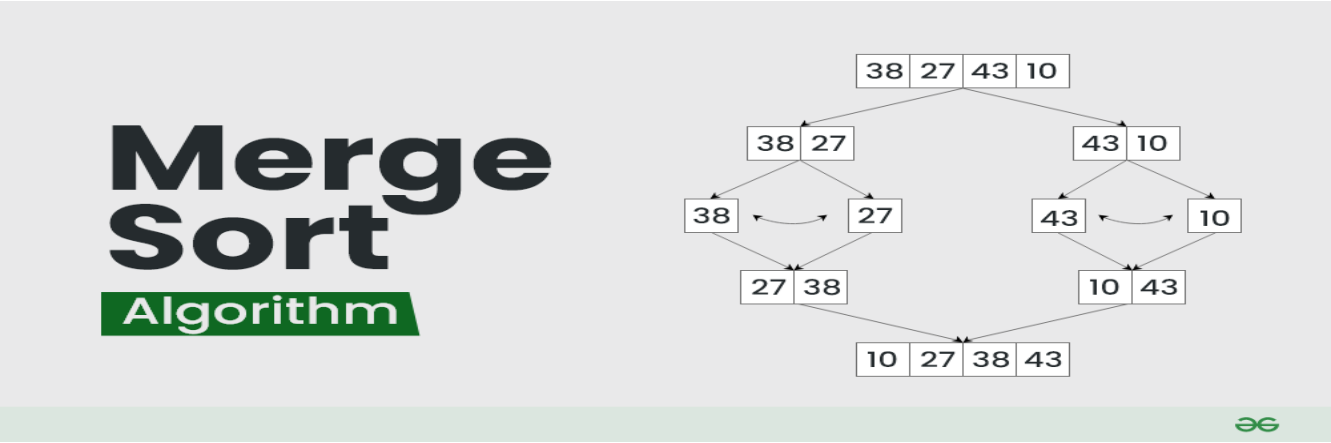
Output

Maximum number of applicants matching for job: 5

3. **Merge Sort**

Merge sort is a sorting algorithm that follows the **divide-and-conquer** approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.

In simple terms, we can say that the process of **merge sort** is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.



Merge Sort Algorithm

How does Merge Sort work?

Merge sort is a popular sorting algorithm known for its efficiency and stability. It follows the **divide-and-conquer** approach to sort a given array of elements.

Here's a step-by-step explanation of how merge sort works:

1. **Divide:** Divide the list or array recursively into two halves until it can no more be divided.
2. **Conquer:** Each subarray is sorted individually using the merge sort algorithm.
3. **Merge:** The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.

Illustration of Merge Sort:

Let's sort the array or list **[38, 27, 43, 10]** using Merge Sort

1 / 4

Let's look at the working of above example:

Divide:

- ***[38, 27, 43, 10]*** *is divided into* ***[38, 27]*** *and* ***[43, 10]*** .
- ***[38, 27]*** *is divided into* ***[38]*** *and* ***[27]*** .
- ***[43, 10]*** *is divided into* ***[43]*** *and* ***[10]*** .

Conquer:

- ***[38]*** *is already sorted.*
- ***[27]*** *is already sorted.*
- ***[43]*** *is already sorted.*
- ***[10]*** *is already sorted.*

Merge:

- *Merge* ***[38]*** *and* ***[27]*** *to get* ***[27, 38]*** .
- *Merge* ***[43]*** *and* ***[10]*** *to get* ***[10,43]*** .
- *Merge* ***[27, 38]*** *and* ***[10,43]*** *to get the final sorted list* ***[10, 27, 38, 43]***

Therefore, the sorted list is ***[10, 27, 38, 43]*** .

Recommended Problem

Merge Sort

.

Implementation of Merge Sort:

C++CJavaPythonC#JavaScriptPHP

```
#include <bits/stdc++.h>
using namespace std;

// Merges two subarrays of arr[].
// First subarray is arr[left..mid]
// Second subarray is arr[mid+1..right]
void merge(vector<int>& arr, int left,
                int mid, int right)
{
   int n1 = mid - left + 1;
```

```
    int n2 = right - mid;

    // Create temp vectors
    vector<int> L(n1), R(n2);

    // Copy data to temp vectors L[] and R[]
    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    int i = 0, j = 0;
    int k = left;

    // Merge the temp vectors back
    // into arr[left..right]
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy the remaining elements of L[],
    // if there are any
    while (i < n1) {
        arr[k] = L[i];
        i++;
```

```
        k++;
    }

    // Copy the remaining elements of R[],
    // if there are any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// begin is for left index and end is right index
// of the sub-array of arr to be sorted
void mergeSort(vector<int>& arr, int left, int right)
{
    if (left >= right)
        return;

    int mid = left + (right - left) / 2;
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);
    merge(arr, left, mid, right);
}

// Function to print a vector
void printVector(vector<int>& arr)
{
    for (int i = 0; i < arr.size(); i++)
        cout << arr[i] << " ";
    cout << endl;
}
```

```
// Driver code
int main()
{
    vector<int> arr = { 12, 11, 13, 5, 6, 7 };
    int n = arr.size();

    cout << "Given vector is \n";
    printVector(arr);

    mergeSort(arr, 0, n - 1);

    cout << "\nSorted vector is \n";
    printVector(arr);
    return 0;
}
```

Output

Given array is

12 11 13 5 6 7

Sorted array is

5 6 7 11 12 13

Recurrence Relation of Merge Sort:

The recurrence relation of merge sort is: $T(n)=\{\Theta(1)\text{if } n=1 2T(n2)+\Theta(n)\text{if } n>1 T(n)=\{\Theta(1)\text{if } n=1 2T(2n)+\Theta(n)\text{if } n>1$

- T(n) Represents the total time time taken by the algorithm to sort an array of size n.
- 2T(n/2) represents time taken by the algorithm to recursively sort the two halves of the array. Since each half has n/2 elements, we have two recursive calls with input size as (n/2).
- O(n) represents the time taken to merge the two sorted halves

Complexity Analysis of Merge Sort:

- **Time Complexity:**
  - **Best Case:** O(n log n), When the array is already sorted or nearly sorted.
  - **Average Case:** O(n log n), When the array is randomly ordered.

- **Worst Case:** O(n log n), When the array is sorted in reverse order.
- **Auxiliary Space:** O(n), Additional space is required for the temporary array used during merging.

Applications of Merge Sort:

- Sorting large datasets
- External sorting (when the dataset is too large to fit in memory)
- Inversion counting
- Merge Sort and its variations are used in library methods of programming languages. For example its variation TimSort is used in Python, Java Android and Swift. The main reason why it is preferred to sort non-primitive types is stability which is not there in QuickSort. For example Arrays.sort in Java uses QuickSort while Collections.sort uses MergeSort.
- It is a preferred algorithm for sorting Linked lists.
- It can be easily parallelized as we can independently sort subarrays and then merge.
- The merge function of merge sort to efficiently solve the problems like union and intersection of two sorted arrays.

Advantages of Merge Sort:

- **Stability** : Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.
- **Guaranteed worst-case performance:** Merge sort has a worst-case time complexity of **O(N logN)** , which means it performs well even on large datasets.
- **Simple to implement:** The divide-and-conquer approach is straightforward.
- **Naturally Parallel** : We independently merge subarrays that makes it suitable for parallel processing.

Disadvantages of Merge Sort:

- **Space complexity:** Merge sort requires additional memory to store the merged sub-arrays during the sorting process.
- **Not in-place:** Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern.
- **Slower than QuickSort in general**. QuickSort is more cache friendly because it works in-place.

NP Class

The NP in NP class stands for **Non-deterministic Polynomial Time**. It is the collection of decision problems that can be solved by a non-deterministic machine in polynomial time.

Features:

- The solutions of the NP class are hard to find since they are being solved by a non-deterministic machine but the solutions are easy to verify.
- Problems of NP can be verified by a Turing machine in polynomial time.

Example:

Let us consider an example to better understand the **NP class**. Suppose there is a company having a total of **1000** employees having unique employee **IDs**. Assume that there are **200** rooms available for them. A selection of **200** employees must be paired together, but the CEO of the company has the data of some employees who can't work in the same room due to personal reasons.

This is an example of an **NP** problem. Since it is easy to check if the given choice of **200** employees proposed by a coworker is satisfactory or not i.e. no pair taken from the coworker list appears on the list given by the CEO. But generating such a list from scratch seems to be so hard as to be completely impractical.

It indicates that if someone can provide us with the solution to the problem, we can find the correct and incorrect pair in polynomial time. Thus for the **NP** class problem, the answer is possible, which can be calculated in polynomial time.

This class contains many problems that one would like to be able to solve effectively:

1. **Boolean Satisfiability Problem (SAT).**

2-Satisfiability (2-SAT) Problem

-
-
-

Boolean Satisfiability Problem

Boolean Satisfiability or simply **SAT** is the problem of determining if a Boolean formula is satisfiable or unsatisfiable.

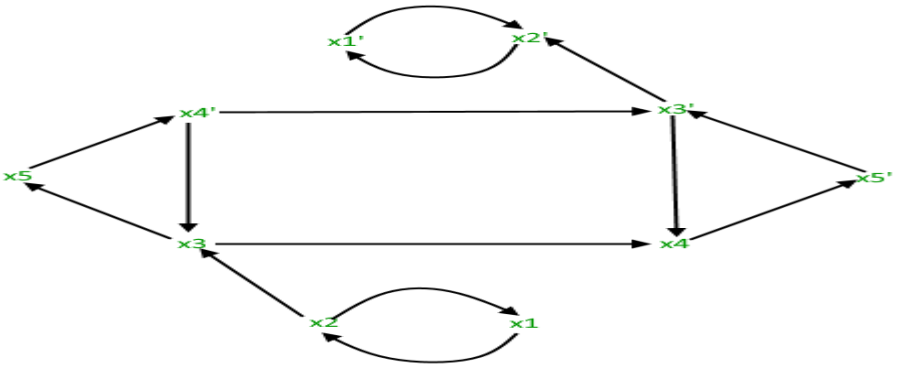
- **Satisfiable :** If the Boolean variables can be assigned values such that the formula turns out to be TRUE, then we say that the formula is satisfiable.
- **Unsatisfiable :** If it is not possible to assign such values, then we say that the formula is unsatisfiable.

Examples:

- F=A∧B¯ $F=A \wedge B^{-}$, is satisfiable, because A = TRUE and B = FALSE makes F = TRUE.
- G=A∧A¯ $G=A \wedge A^{-}$, is unsatisfiable, because:

A*A*	A¯*A*¯	G*G*
TRUE	**FALSE**	**FALSE**
FALSE	**TRUE**	**FALSE**

Note : Boolean satisfiability problem is NP-complete (For proof, refer Cook's Theorem).



What is 2-SAT Problem

2-SAT is a special case of Boolean Satisfiability Problem and can be solved in polynomial time.

To understand this better, first let us see what is Conjunctive Normal Form (CNF) or also known as Product of Sums (POS).

CNF : CNF is a conjunction (AND) of clauses, where every clause is a disjunction (OR). Now, 2-SAT limits the problem of SAT to only those Boolean formula which are expressed as a CNF with every clause having only **2 terms**(also called **2-CNF**).

Example: F=(A1∨B1)∧(A2∨B2)∧(A3∨B3)∧…….∧(Am∨Bm) $F=(A1 \vee B1) \wedge (A2 \vee B2) \wedge (A3 \vee B3) \wedge \ldots\ldots\ldots \wedge (Am \vee Bm)$

Thus, Problem of 2-Satisfiability can be stated as:

Given CNF with each clause having only 2 terms, is it possible to assign such values to the variables so that the CNF is TRUE?

Examples:

Input : F=(x1∨x2)∧(x2∨x1¯)∧(x1¯∨x2¯)F=(x1∨x2)∧(x2∨x1¯)∧(x1¯∨x2¯)Output : The given expression is satisfiable. (for x1 = FALSE, x2 = TRUE)Input : F=(x1∨x2)∧(x2∨x1¯)∧(x1∨x2¯)∧(x1¯∨x2¯)F=(x1∨x2)∧(x2∨x1¯)∧(x1∨x2¯)∧(x1¯∨x2¯)Output : The given expression is unsatisfiable. (for all possible combinations of x1 and x2)

Approach for 2-SAT Problem

For the CNF value to come TRUE, value of every clause should be TRUE. Let one of the clause be (A∨B) $(A \vee B)$. **= TRUE**

- If A = 0, B must be 1 i.e. $(A^{-} \Rightarrow B)$ $(A^{-} \Rightarrow B)$
- If B = 0, A must be 1 i.e. $(B^{-} \Rightarrow A)$ $(B^{-} \Rightarrow A)$

Thus,

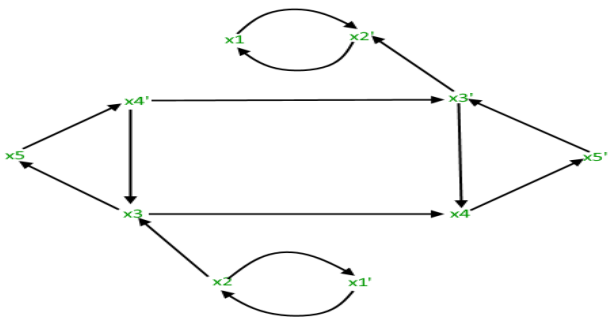
(A∨B)$(A \vee B)$ = TRUE is equivalent to $(A^{-} \Rightarrow B) \wedge (B^{-} \Rightarrow A)$ $(A^{-} \Rightarrow B) \wedge (B^{-} \Rightarrow A)$

Now, we can express the CNF as an Implication. So, we create an Implication Graph which has 2 edges for every clause of the CNF. (A∨B) $(A \vee B)$ is expressed in Implication Graph as [Tex]edge(\bar{A} \rightarrow B) \ & edge(\bar{B} \rightarrow A) [/Tex]

Thus, for a Boolean formula with 'm' clauses, we make an Implication Graph with:

- 2 edges for every clause i.e. '2m' edges.
- 1 node for every Boolean variable involved in the Boolean formula.

Let's see one example of Implication Graph.



Note: The implication (if A then B) is equivalent to its contrapositive (if B¯ *B*¯ then A¯ *A*¯).

Now, consider the following cases:

CASE 1: If *edge(X→X¯)edge(X→X¯)existsinthegraphexistsinthegraphThis means (X⇒X¯)(X⇒X¯)If X = TRUE, X¯X¯ = TRUE, which is a contradiction.But if X = FALSE, there are no implication constraints.Thus, X = FALSE*

CASE 2: If *edge(X¯→X)edge(X¯→X)existsinthegraphexistsinthegraphThis means (X¯⇒X)(X¯⇒X)If X¯X¯ = TRUE, X = TRUE, which is a contradiction.But if X¯X¯ = FALSE, there are no implication constraints.Thus, X¯X¯ = FALSE i.e. X = TRUE*

CASE 3: If *edge(X→X¯)&edge(X¯→X)edge(X→X¯)&edge(X¯→X)bothexistinthegraphbothexistinthegraphOne edge requires X to be TRUE and the other one requires X to be FALSE.Thus, there is no possible assignment in such a case.*

2. **Hamiltonian Path Problem.**

Proof that Hamiltonian Cycle is NP-Complete

-
-
-

Prerequisite: NP-Completeness, Hamiltonian cycle.

Hamiltonian Cycle:

A cycle in an undirected graph G=(V, E) traverses every vertex exactly once.

Problem Statement: Given a graph G=(V, E), the problem is to determine if graph G contains a Hamiltonian cycle consisting of all the vertices belonging to V.

Explanation: An instance of the problem is an input specified to the problem. An instance of the Independent Set problem is a graph G=(V, E), and the problem is to check whether the graph can have a Hamiltonian Cycle in G. Since an NP-Complete problem, by definition, is a problem which is both in NP and NP-hard, the proof for the statement that a problem is NP-Complete consists of two parts:

1. *The problem itself is in NP class.*
2. *All other problems in NP class can be polynomial-time reducible to that. (B is polynomial-time reducible to C is denoted as)*

If the 2nd condition is only satisfied then the problem is called **NP-Hard**. But it is not possible to reduce every NP problem into another NP problem to show its NP-Completeness all the time. That is why if we want to show a problem is NP-Complete, we just show that the problem is in **NP** and if any **NP-Complete** problem is reducible to that, then we are done, i.e. if B is NP-Complete and for C in NP, then C is NP-Complete.

1. **Hamiltonian Cycle is in NP** If any problem is in NP, then, given a *'certificate'*, which is a solution to the problem and an instance of the problem (a graph G and a positive integer k, in this case), we will be able to verify (check whether the solution given is correct or not) the certificate in polynomial time. The certificate is a sequence of vertices forming Hamiltonian Cycle in the graph. We can validate this solution by verifying that all the vertices belong to the graph and each pair of vertices belonging to the solution are adjacent. This can be done in polynomial time, that is **O(V +E)** using the following strategy for graph G(V, E):

   flag=true

   For every pair {u, v} in the subset V':

   Check that these two have an edge between them

   If there is no edge, set flag to false and break

   If flag is true:

   Solution is correct

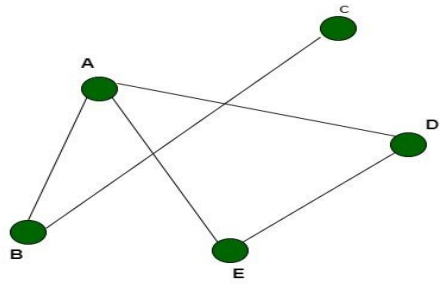
   Else:

   Solution is incorrect

1. **Hamiltonian Cycle is NP Hard** In order to prove the Hamiltonian Cycle is NP-Hard, we will have to reduce a known NP-Hard problem to this problem. We will carry out a reduction from the Hamiltonian Path problem to the Hamiltonian Cycle problem. Every instance of the Hamiltonian Path problem consisting of a graph **G =(V, E)** as the input can be converted to Hamiltonian Cycle problem consisting of graph **G' = (V', E')**. We will construct the graph G' in the following way:

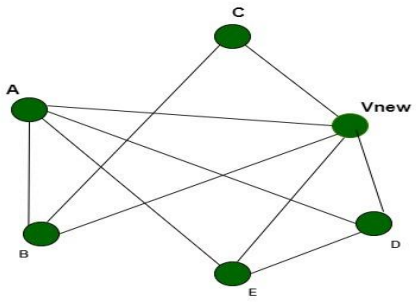
- **V'** = Add vertices V of the original graph G and add an additional vertex $\mathbf{V_{new}}$ such that all the vertices connected of the graph are connected to this vertex. The number of vertices increases by 1, **V' =V+1**.

- **E'** = Add edges E of the original graph G and add new edges between the newly added vertex and the original vertices of the graph. The number of edges increases by the number of vertices V, that is, **E'=E+V**.
- Let us assume that the graph G contains a hamiltonian path covering the **V** vertices of the graph starting at a random vertex say $\mathbf{V_{start}}$ and ending at Vend, now since we connected all the vertices to an arbitrary new vertex $\mathbf{V_{new}}$ in G'. We extend the original Hamiltonian Path to a Hamiltonian Cycle by using the edges $\mathbf{V_{end}}$ to $\mathbf{V_{new}}$ and V_{new} to V_{start} respectively. The graph **G'** now contains the closed cycle traversing all vertices once.
- We assume that the graph **G'** has a *Hamiltonian Cycle* passing through all the vertices, inclusive of $\mathbf{V_{new}}$. Now to convert it to a *Hamiltonian Path*, we remove the edges corresponding to the vertex $\mathbf{V_{new}}$ in the cycle. The resultant path will cover the vertices V of the graph and will cover them exactly once.



G ≅

Hamiltonian Parh DEABC



G' ≅

Hamiltonian Cycle DEABCVnewD

Thus we can say that the graph **G'** contains a *Hamiltonian Cycle* if graph **G** contains a *Hamiltonian Path*. Therefore, any instance of the *Hamiltonian Cycle* problem can be reduced to an instance of the *Hamiltonian Path* problem. Thus, the *Hamiltonian Cycle* is **NP-Hard**. **Conclusion:** Since, the *Hamiltonian Cycle* is both, a **NP-Problem** and **NP-Hard**. Therefore, it is a **NP-Complete** problem.

3. **Graph coloring.**

Proof that vertex cover is NP complete

-
-
-

Prerequisite – Vertex Cover Problem, NP-Completeness

Problem – Given a graph G(V, E) and a positive integer k, the problem is to find whether

there is a subset V' of vertices of size at most k, such that every edge in the graph is connected to some vertex in V'.

Explanation –

First let us understand the notion of an instance of a problem. An instance of a problem is nothing but an input to the given problem. An instance of the Vertex Cover problem is a graph G (V, E) and a positive integer k, and the problem is to check whether a vertex cover of size at most k exists in G. Since an NP Complete problem, by definition, is a problem which is both in NP and NP hard, the proof for the statement that a problem is NP Complete consists of two parts:

1. **Proof that vertex cover is in NP –**

   If any problem is in NP, then, given a 'certificate' (a solution) to the problem and an instance of the problem (a graph G and a positive integer k, in this case), we will be able to verify (check whether the solution given is correct or not) the certificate in polynomial time.

   The certificate for the vertex cover problem is a subset V' of V, which contains the vertices in the vertex cover. We can check whether the set V' is a vertex cover of size k using the following strategy (for a graph G(V, E)):

```
let count be an integer
set count to 0
for each vertex v in V'
   remove all edges adjacent to v from set E
   increment count by 1
   if count = k and E is empty
   then
      the given solution is correct
   else
      the given solution is wrong
```

   It is plain to see that this can be done in polynomial time. Thus the vertex cover problem is in the class NP.

2. **Proof that vertex cover is NP Hard –**

   To prove that Vertex Cover is NP Hard, we take some problem which has already been proven to be NP Hard, and show that this problem can be reduced to the Vertex Cover

problem. For this, we consider the Clique problem, which is NP Complete (and hence NP Hard).

"In computer science, the clique problem is the computational problem of finding cliques (subsets of vertices, all adjacent to each other, also called complete subgraphs) in a graph."

Here, we consider the problem of finding out whether there is a clique of size k in the given graph. Therefore, an instance of the clique problem is a graph G (V, E) and a non-negative integer k, and we need to check for the existence of a clique of size k in G.

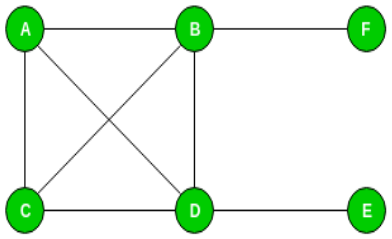
Now, we need to show that any instance (G, k) of the Clique problem can be reduced to an instance of the vertex cover problem. Consider the graph G' which consists of all edges not in G, but in the complete graph using all vertices in G. Let us call this the complement of G. Now, the problem of finding whether a clique of size k exists in the graph G is the same as the problem of finding whether there is a vertex cover of size |V| – k in G'. We need to show that this is indeed the case.

Assume that there is a clique of size k in G. Let the set of vertices in the clique be V'. This means |V'| = k. In the complement graph G', let us pick any edge (u, v). Then at least one of u or v must be in the set V – V'. This is because, if both u and v were from the set V', then the edge (u, v) would belong to V', which, in turn would mean that the edge (u, v) is in G. This is not possible since (u, v) is not in G. Thus, all edges in G' are covered by vertices in the set V – V'.

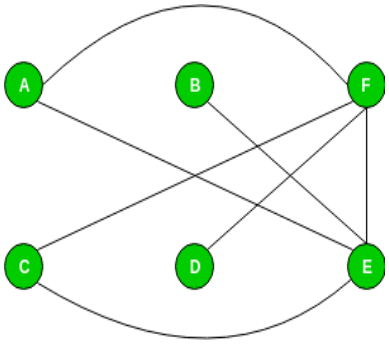
Now assume that there is a vertex cover V'' of size |V| – k in G'. This means that all edges in G' are connected to some vertex in V''. As a result, if we pick any edge (u, v) from G', both of them cannot be outside the set V''. This means, all edges (u, v) such that both u and v are outside the set V'' are in G, i.e., these edges constitute a clique of size k.

Thus, we can say that there is a clique of size k in graph G if and only if there is a vertex cover of size |V| – k in G', and hence, any instance of the clique problem can be reduced to an instance of the vertex cover problem. Thus, vertex cover is NP Hard. Since vertex cover is in both NP and NP Hard classes, it is NP Complete.

To understand the proof, consider the following example graph and its complement:



G
V' ={A, B, C, D}



G'
V" ={E, F}

Co-NP Class

Co-NP stands for the complement of NP Class. It means if the answer to a problem in Co-NP is No, then there is proof that can be checked in polynomial time.

Features:

- If a problem X is in NP, then its complement X' is also in CoNP.
- For an NP and CoNP problem, there is no need to verify all the answers at once in polynomial time, there is a need to verify only one particular answer "yes" or "no" in polynomial time for a problem to be in NP or CoNP.

Some example problems for CoNP are:

1. **To check prime number.**

Prime Numbers

-

Prime numbers are those **natural numbers divisible by only 1 and the number itself**. Numbers with more than two factors are called composite numbers. All primes are odd numbers except for 2.

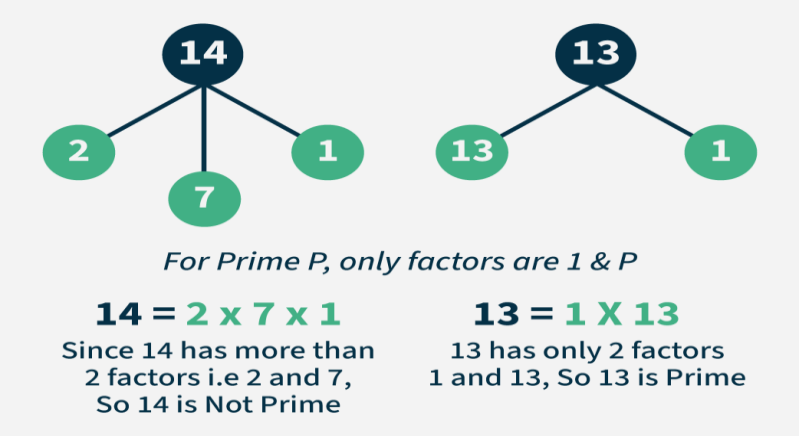
A total of 25 prime numbers are there between 1 and 100. These are: ***2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97.***

Mathematically prime numbers are defined as:

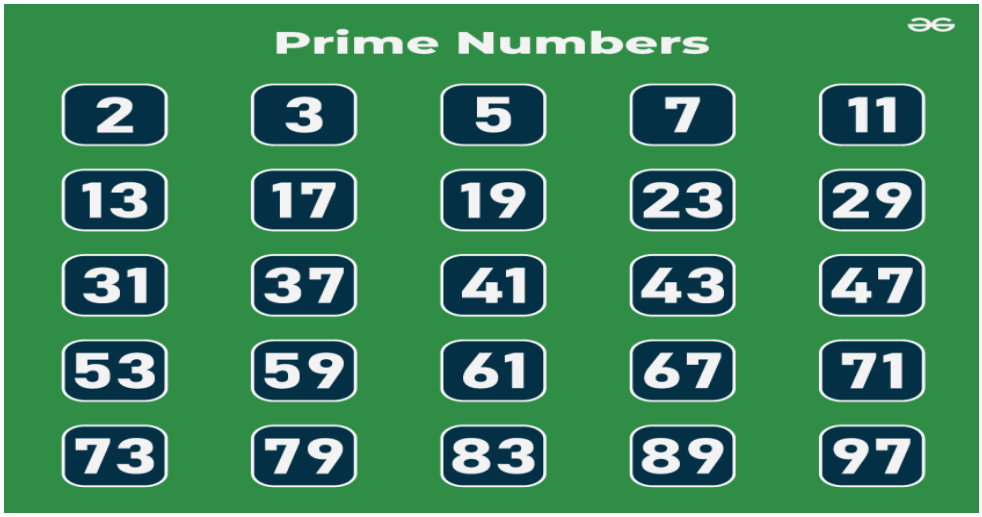
If $p \in N$, then p is prime if for any integers a and b, if $p = a \cdot b$, then either $a = 1$ or $b = 1$

OR

p is prime $\Leftrightarrow p > 1$ and if $p = a \cdot b$, then either $a = 1$ or $b = 1$.



There are 25 prime numbers between 1 to 100. The image added below shows the prime numbers from 1 to 100.



Smallest and Largest Prime Number

The **smallest prime number** is 2. It is unique because it's the only even prime number. There is **no largest prime number**, as prime numbers continue infinitely

Read More: Smallest and Largest Prime Number

Properties of Prime Numbers

Prime numbers have some properties which are listed as follows:

- A Prime number is always a natural number i.e., a number greater than 1.
- There are infinitely many prime numbers. This was proved by Euclid around 300 BC.
- Every positive integer greater than 1 can be written as a product of prime numbers in a unique way, up to the order of the factors. This is known as the Fundamental Theorem of Arithmetic.

Please refer **Interesting Facts about Prime Numbers** for more interesting facts.

Prime Factorization

Prime factorization involves breaking down a composite number into a product of prime numbers. For example, the prime factorization of 60 is 22 x 3 x 5. This is useful for solving problems involving divisibility, greatest common divisors, and least common multiples.

Prime Factorization of Some Numbers

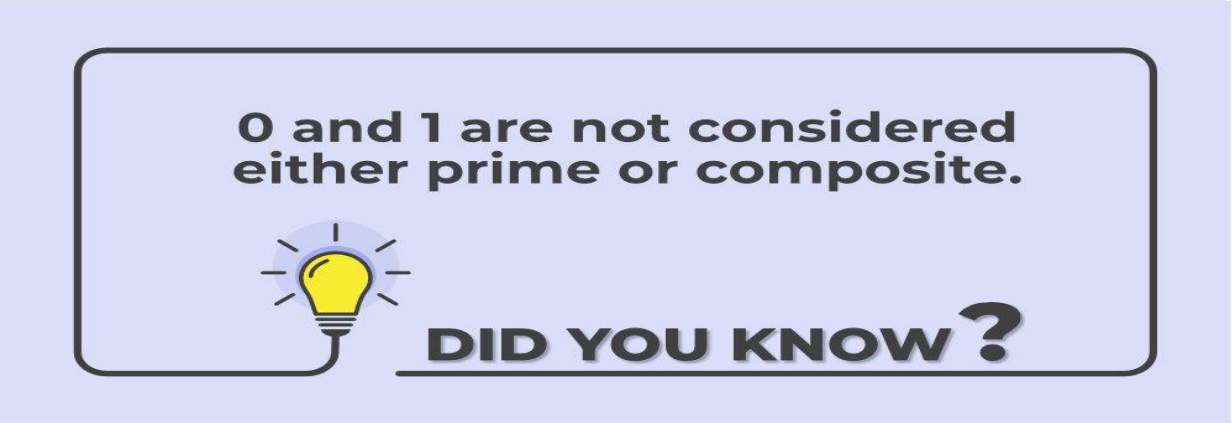
Prime factorization of some of the common composite number are:

Numbers	Prime Factorization	Numbers	Prime Factorization
36	22 × 32	40	23 × 5
24	23 × 3	50	2 × 52
60	22 × 3 × 5	48	24 × 3
18	2 × 32	30	2 × 3 × 5
72	23 × 32	42	2 × 3 × 7
45	32 × 5	20	22 × 5

Prime Numbers vs Composite Numbers

There are a few differences between Prime and Composite numbers, which are as follows:

Prime Numbers	Composite Numbers
All Prime Numbers have only two factors such as 1 and the number itself. For example, 5 has only two factors 1 and 5.	All Composites Numbers must have any factor other than 1 and the number itself. For example, 6 has 2 and 3 as its factors other than 1 and 6.
All Primes are Odd numbers except 2.	Composites can be either odd or even.
Some of the first few primes are 2, 3, 5, 7, 11, 13, 17, etc.	Some of the first few composites are 4, 6, 8, 9, 12, 14, 15, etc.



Coprime Numbers

Coprime numbers have just a single common factor, which is 1. This means that a pair of numbers is said to be coprime if their largest common factor is 1.

Note: *All primes numbers are coprime to each other.*

Read More: Co Prime Numbers

Interesting Facts about Prime Numbers

Some of the interesting facts about Prime Numbers are mentioned below:

- With the exception of 2, which is an even prime number, all prime numbers are odd.
- All prime numbers, with the exception of 2 and 5, ends with 1, 3, 7, or 9.
- A product of two or more prime numbers can be used to express any positive number larger than one in a unique way.
- Two prime integers can be added to represent any even positive integer.

How to Find Prime Numbers?

Let's learn some simple methods, which can help us easily identify that given number is a Prime or not.

- All even numbers are NOT Prime, except for 2. This means if a number ends with 0, 2, 4, 6, or 8, it is not prime.
- Prime Numbere can also be found by Divisibility Rules for Small Primes.
- Division Method and Factorization Method are also used to find prime numbers.

Algorithms to Check for Prime

Since there is no formula to directly check whether a given number is prime or not, we typically use compute programs to check the same. There exist many algorithms to do the same. The best possible time we get for prime numbers is O(Log n) for a given number n if we have not done any preconceptions. Please refer Check for Prime Number for all programs and algorithms in detail.

Applications of Prime Numbers in Real Life

There are so many real life examples of prime numbers in real life. These uses and applications are mentioned below:

- Cryptography
- RSA Encryption
- Internet Security
- Digital Signatures
- Credit Card Transactions

Theorems related to Prime Numbers

Some of the important theorems related to prime numbers are:

- Euclid Division Lemma
- Wilson Theorem
- Fermat's Little Theorem

Euclid Division Lemma

Euclid's Division Lemma gives the relation between the various components of Division. It explains that for any two positive integers say a and b there exist two unique integers q and r such that a = bq + r where q the quotient of the division, and r is the remainder of the division.

Wilson Theorem

Wilson Theorem states that if p is any natural number greater than 1, then p is said to be a prime number if and only if the product of all the positive integers less than p is one less than a multiple of number p.

Fermat's Little Theorem

According to Fermat's Little theorem, if p is a prime number and a is a positive integer not divisible by p, then there exists a positive integer k such that ak – 1 is divisible by p.

Importance of Prime Numbers

Prime Numbers hold a significant place in the evolution of mathematics, primarily for their crucial role in improving cryptography and security standards in the modern world. Despite years of research, Prime Numbers continue to be an active topic in the field of mathematical research, fascinating scientists and scholars with their elegant properties and wide range of applications.

- Prime numbers are the basic building blocks of natural numbers. Every integer greater than 1 can be uniquely expressed as a product of primes, known as the Fundamental Theorem of Arithmetic. This uniqueness is crucial for number theory.
- Prime numbers play a vital role in modern cryptography, particularly in algorithms like RSA. They are used to create keys for secure data transmission, ensuring that sensitive information remains private.
- Primes are central to many important theorems and conjectures in mathematics, such as the Goldbach Conjecture and the Prime Number Theorem, which describes the distribution of prime numbers among the integers.
- In computer algorithms, prime numbers are used in hashing functions and data structures, helping to optimize performance and reduce collisions in hash tables.
- Studying prime numbers contributes to the understanding of randomness and patterns in mathematics. Their distribution can reveal insights into more complex mathematical structures and theories.

Prime Numbers Examples

Example 1: Determine whether 37 is a prime number or not.

Solution:

As a prime number is a positive integer greater than 1 that has no positive integer divisors other than 1 and itself.

As 37 = 1 × 37,

Therefore, 37 has no divisors other than 1 and itself, and it is a prime number. So, 37 is a prime number.

Example 2: Find all the prime numbers between 20 and 40.

Solution:

The prime numbers between 20 and 40 are 23, 29, and 37.

Example 3: Is 19 a Prime Number?

Solution:

Let us write the given number in the form of 6n ± 1.

6(3) + 1 = 18 + 1 = 19

Therefore, 19 is a prime number

Example 4: Which is the greatest prime number between 1 to 10?

Solution:

There are 4 prime numbers between 1 and 10 and the greatest prime number between 1 and 10 is 7.

Examples 5: Why is 20 not a prime number?

Solution:

The factors of 20 are 1, 2, 4, 5, 10, and 20. Thus, 20 has more than two factors. Since the number of factors of 20 is more than two numbers, it is NOT a prime number.

2. **Integer Factorization.**

-
-

Given a number **n**, write an efficient function to print all prime factors of **n**. For example, if the input number is 12, then the output should be “2 2 3”. And if the input number is 315, then the output should be “3 3 5 7”.

First Approach:

Following are the steps to find all prime factors.

1) While n is divisible by 2, print 2 and divide n by 2.

2) After step 1, n must be odd. Now start a loop from i = 3 to the square root of n. While i divides n, print i, and divide n by i. After i fails to divide n, increment i by 2 and continue.

3) If n is a prime number and is greater than 2, then n will not become 1 by the above two steps. So print n if it is greater than 2.

Recommended Problem

Largest prime factor

```
// C++ program to print all prime factors
#include <bits/stdc++.h>
using namespace std;

// A function to print all prime
// factors of a given number n
void primeFactors(int n)
{
    // Print the number of 2s that divide n
    while (n % 2 == 0)
    {
        cout << 2 << " ";
        n = n/2;
    }

    // n must be odd at this point. So we can skip
    // one element (Note i = i +2)
    for (int i = 3; i <= sqrt(n); i = i + 2)
    {
        // While i divides n, print i and divide n
        while (n % i == 0)
        {
            cout << i << " ";
            n = n/i;
        }
    }

    // This condition is to handle the case when n
    // is a prime number greater than 2
    if (n > 2)
        cout << n << " ";
```

```
}

/* Driver code */
int main()
{
   int n = 315;
   primeFactors(n);
   return 0;
}

// This is code is contributed by rathbhupendra
```

Output

3 3 5 7

Time Complexity: O(sqrt(n))

In the worst case (when either n or sqrt(n) is prime, for example: take n=11 or n=121 for both the cases for loop runs sqrt(n) times), the for loop runs for sqrt(n) times. The more number of times the while loop iterates on a number it reduces the original n, which also reduces the value of sqrt(n). Although the best case time complexity is O(log(n)), when the prime factors of n is only 2 and 3 or n is of the form (2^x*(3^y) where x>=0 and y>=0.

Auxiliary Space: O(1)

How does this work?

The steps 1 and 2 take care of composite numbers and step 3 takes care of prime numbers. To prove that the complete algorithm works, we need to prove that steps 1 and 2 actually take care of composite numbers. This is clear that step 1 takes care of even numbers. And after step 1, all remaining prime factors must be odd (difference of two prime factors must be at least 2), this explains why i is incremented by 2.

Now the main part is, the loop runs till the square root of n not till n. To prove that this optimization works, let us consider the following property of composite numbers.

Every composite number has at least one prime factor less than or equal to the *square root of itself.*

This property can be proved using a counter statement. Let a and b be two factors of n such

that a*b = n. If both are greater than ?n, then a.b > ?n, * ?n, which contradicts the expression "a * b = n".

In step 2 of the above algorithm, we run a loop and do the following in loop

a) Find the least prime factor i (must be less than ?n,)

b) Remove all occurrences i from n by repeatedly dividing n by i.

c) Repeat steps a and b for divided n and i = i + 2. The steps a and b are repeated till n becomes either 1 or a prime number.

NP-hard class

An NP-hard problem is at least as hard as the hardest problem in NP and it is a class of problems such that every problem in NP reduces to NP-hard.

Features:

- All NP-hard problems are not in NP.
- It takes a long time to check them. This means if a solution for an NP-hard problem is given then it takes a long time to check whether it is right or not.
- A problem A is in NP-hard if, for every problem L in NP, there exists a polynomial-time reduction from L to A.

Some of the examples of problems in Np-hard are:

1. **Halting problem.**
2. **Qualified Boolean formulas.**
3. **No Hamiltonian cycle.**

NP-complete class

A problem is NP-complete if it is both NP and NP-hard. NP-complete problems are the hard problems in NP.

Features:

- NP-complete problems are special as any problem in NP class can be transformed or reduced into NP-complete problems in polynomial time.
- If one could solve an NP-complete problem in polynomial time, then one could also solve any NP problem in polynomial time.

Some example problems include:

1. **Hamiltonian Cycle.**
2. **Satisfiability.**
3. **Vertex cover.**

Complexity Class	Characteristic feature
P	Easily solvable in polynomial time.
NP	Yes, answers can be checked in polynomial time.
Co-NP	No, answers can be checked in polynomial time.
NP-hard	All NP-hard problems are not in NP and it takes a long time to check them.
NP-complete	A problem that is NP and NP-hard is NP-complete.

Cook's Theorem

Stephen Cook presented four theorems in his paper "The Complexity of Theorem Proving Procedures". These theorems are stated below. We do understand that many unknown terms are being used in this chapter, but we don't have any scope to discuss everything in detail.

Following are the four theorems by Stephen Cook −

Theorem-1

If a set S of strings is accepted by some non-deterministic Turing machine within polynomial time, then S is P-reducible to {DNF tautologies}.

Theorem-2

The following sets are P-reducible to each other in pairs (and hence each has the same polynomial degree of difficulty): {tautologies}, {DNF tautologies}, D3, {sub-graph pairs}.

Theorem-3

For any TQ(k) of type Q, TQ(k)k√(logk)2
is unbounded

There is a TQ(k) of type Q such that TQ(k)⩽2k(logk)2

Theorem-4

If the set S of strings is accepted by a non-deterministic machine within time T(n) = 2n, and if TQ(k) is an honest (i.e. real-time countable) function of type Q, then there is a constant K, so S can be recognized by a deterministic machine within time TQ(K8n).

First, he emphasized the significance of polynomial time reducibility. It means that if we have a polynomial time reduction from one problem to another, this ensures that any polynomial time algorithm from the second problem can be converted into a corresponding polynomial time algorithm for the first problem

Second, he focused attention on the class NP of decision problems that can be solved in polynomial time by a non-deterministic computer. Most of the intractable problems belong to this class, NP.

Third, he proved that one particular problem in NP has the property that every other problem in NP can be polynomially reduced to it. If the satisfiability problem can be solved with a polynomial time algorithm, then every problem in NP can also be solved in polynomial time. If any problem in NP is intractable, then satisfiability problem must be intractable. Thus, satisfiability problem is the hardest problem in NP.

Fourth, Cook suggested that other problems in NP might share with the satisfiability problem this property of being the hardest member of NP.

NP-complete problem, any of a class of computational problems for which no efficient solution algorithm has been found. Many significant computer-science problems belong to this class—e.g., the traveling salesman problem, satisfiability problems, and graph-covering problems.

So-called easy, or tractable, problems can be solved by computer algorithms that run in polynomial time; i.e., for a problem of size *n*, the time or number of steps needed to find the solution is a polynomial function of *n*. Algorithms for solving hard, or intractable, problems, on the other hand, require times that are exponential functions of the problem size *n*. Polynomial-time algorithms are considered to be efficient, while exponential-time algorithms are considered inefficient, because the execution times of the latter grow much more rapidly as the problem size increases.

A problem is called NP (nondeterministic polynomial) if its solution can be guessed and verified in polynomial time; nondeterministic means that no particular rule is followed to make the guess. If a problem is NP and all other NP problems are polynomial-time reducible to it, the problem is NP-complete. Thus, finding an efficient algorithm for any NP-complete problem implies that an efficient algorithm can be found for all such problems, since any problem belonging to this class can be recast into any other member of the class. It is not known whether any polynomial-time algorithms will ever be found for NP-complete problems, and determining whether these problems are tractable or intractable remains one of the most important questions in theoretical computer science. When an NP-complete problem must be solved, one approach is to use a polynomial algorithm to approximate the solution; the answer thus obtained will not necessarily be optimal but will be reasonably close.

Cook–Levin theorem or Cook's theorem

In computational complexity theory, the Cook–Levin theorem, also known as Cook's theorem, states that the Boolean satisfiability problem is NP-complete. That is, it is in NP, and any problem in NP can be reduced in polynomial time by a deterministic Turing machine to the Boolean satisfiability problem.

Stephen Arthur Cook and L.A. Levin in 1973 independently proved that the ***satisfiability problem(SAT)*** *is NP-complete. Stephen Cook, in 1971, published an important paper titled 'The complexity of Theorem Proving Procedures', in which he outlined the way of obtaining the proof of an NP-complete problem by reducing it to* ***SAT****. He proved* ***Circuit-SAT*** *and* ***3CNF-SAT*** *problems are as hard as* ***SAT****. Similarly, Leonid Levin independently worked on this problem in the then Soviet Union. The proof that* ***SAT*** *is NP-complete was obtained due to the efforts of these two scientists. Later, Karp reduced 21 optimization problems, such as Hamiltonian tour, vertex cover, and clique, to the* ***SAT*** *and proved that those problems are NP-complete.*

Hence, an ***SAT*** *is a significant problem and can be stated as follows:*

Given a boolean expression **F** having **n** variables $x_1, x_2, \ldots, x_n$, and Boolean operators, is it possible to have an assignment for variables true or false such that binary expression **F** is true?

This problem is also known as the **formula – SAT**. An **SAT(formula-SAT or simply SAT)** takes a Boolean expression F and checks whether the given expression(or formula) is satisfiable. A Boolean expression is said to be satisfactory for some valid assignments of

variables if the evaluation comes to be true. Prior to discussing the details of SAT, let us now discuss some important terminologies of a Boolean expression.

- **Boolean variable:** A variable, say **x**, that can have only two values, true or false, is called a boolean variable
- **Literal:** A literal can be a logical variable, say **x**, or the negation of it, that is **x** or $\bar{x}$; **x** is called a positive literal, and $\bar{x}$ is called the negative literal
- **Clause:** A sequence of variables$(x_1,x_2,\ldots,x_n)$ that can be separated by a logical **OR** operator is called a clause. For example, $(x_1 \vee x_2 \vee x_3)$ is a clause of three literals.
- **Expressions:** One can combine all the preceding clauses using a Boolean operator to form an expression.
- **CNF form:** An expression is in CNF form(conjunctive normal form) if the set of clauses are separated by an **AND (^),** operator, while the literals are connected by an **OR (v)** operator. The following is an example of an expression in the CNF form:
  - $f = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee \bar{x}_3 \vee x_2)$
- **3 – CNF:** An expression is said to be in 3-CNF if it is in the conjunctive normal form, and every clause has exact three literals.

  Thus, an **SAT** is one of the toughest problems, as there is no known algorithm other than the brute force approach. A brute force algorithm would be an exponential-time algorithm, as 2^n possible assignments need to be tried to check whether the given Boolean expression is true or not. Stephen Cook and Leonid Levin proved that the **SAT** is NP-complete.

  Cook demonstrated the reduction of other hard problems to SATs. Karp provided proof of 21 important problems, Such as Hamiltonian tour, vertex cover, and clique, by reducing it to SAT using Karp reduction.

  Let us briefly introduce the three types of SATs. They are as follows:

1. **Circuit- SAT:** A circuit-SAT can be stated as follows: given a Boolean circuit, which is a collection of gates such as **AND**, **OR**, and **NOT**, and **n** inputs, is there any input assignments of Boolean variables so that the output of the given circuit is true? Again, the difficulty with these problems is that for **n** inputs to the circuit. 2^n possible outputs should be checked. Therefore, this brute force algorithm is an exponential algorithm and hence this is a hard problem.
2. **CNF-SAT:** This problem is a restricted problem of **SAT**, where the expression should be in a conjunctive normal form. An expression is said to be in a conjunction form if all the clauses

are connected by the Boolean **AND** operator. Like in case of a **SAT**, this is about assigning truth values to n variables such that the output of the expression is true**.**

3. **3-CNF-SAT(3-SAT):** This problem is another variant where the additional restriction is that the expression is in a conjunctive normal form and that every clause should contain exactly three literals. This problem is also about assigning **n** assignments of truth values to **n** variables of the Boolean expression such that the output of the expression is true. In simple words, given an expression in 3-CNF, a 3-SAT problem is to check whether the given expression is satisfiable.

What is an NP-complete problem?

the NP-complete problems. At its core, an NP-complete problem is a type of problem for which no efficient solution has been found, but if a solution is given, it can be verified quickly. Imagine you're trying to solve a jigsaw puzzle. Finding the correct arrangement of pieces can be time-consuming. But if someone shows you the completed picture, you can instantly verify it's correct. That's the essence of NP-complete problems: hard to solve, but easy to verify.

Famous NP-complete Problems: A Glimpse into Complexity

The realm of NP-complete problems is vast, with many challenges that have stumped the brightest minds for decades. Let's explore some of the most iconic NP-complete problems that have shaped the course of computational theory:

1. The Traveling Salesman Problem (TSP):

Imagine a salesman who needs to visit multiple cities and return to the starting point, all while minimizing the total distance traveled. While it sounds straightforward, as the number of cities increases, the problem's complexity grows exponentially. Finding the shortest possible route becomes a daunting task. Dive deeper into TSP.

2. The Knapsack Problem:

Given a set of items, each with a weight and a value, determine the number of each item to include in a knapsack so that the total weight doesn't exceed a given limit, and the total value is maximized. It's like packing for a hiking trip, where you want to maximize utility while staying within weight constraints. Learn more about the Knapsack Problem.

3. Boolean Satisfiability Problem (SAT):

Can you find an assignment of truth values to variables that makes a given Boolean formula true? This problem is foundational in computer science and has implications in areas like hardware design and artificial intelligence. Explore the intricacies of SAT.

4. The Clique Problem:

In a social network, can you find a group of people (a clique) where everyone knows everyone else within the group? As the network grows, pinpointing such cliques becomes increasingly complex. Delve into the Clique Problem.

These problems, while diverse in nature, share a common thread: they are all NP-complete. This means that if a polynomial-time solution is found for one, it would exist for all of them, revolutionizing the field of computer science

Class P and NP class of proble ms

P is the set of all decision problems which can be solved in polynomial time by a deterministic Turing machine. A problem is feasible if it has a solution cost as a polynomial. All problems that can be solved in polynomial time is called polynomial time or class P problems. The class of decision problems that can be solved by a non-deterministic polynomial algorithm is called class NP problem.

Class NP problems are hard problems. 'Hard' in the sense, these problems require more computer resources such as CPU time and memory to solve these problems. Also, for most of these problems, no polynomial time algorithms exist for these problems. Also, it can be observed that most of these problems are combinatorial optimization problems and most of the combinatorial problems are hard problems. NP-Complete (or NPC) problems are a set of problems that are well connected. A problem x that is in NP, if known to have a polynomial time algorithm, it implies that polynomial time algorithm exist for all NP-Complete problems.

Reductions

Reduction algorithm reduces a problem to another problem. The input of a problem is called instance. Problem A is reduced to another problem B, if any instance of A "can be rephrased" as instance of B, the solution of which provides a solution to the instance of A [3]. There are two types of reduction. One is called Turing reduction and another is called Karp reduction.

Turing reduction

Let us assume two problems A and B. Problem B has a solution while problem A does not have any solution. Then reduction reduces attempts to solve problem A using procedure of solving problem B. In Turing reduction, problem A is solved using a procedure that solves B. Thus, efficient procedure for Problem A would be the efficient procedure for problem B. It can be shown mathematically denoted as follows:

A £*P* B

This implies that problem B is at least as hard as problem A and also in other words, problem A cannot be harder than problem B.

Karp Reduction

Karp reduction is another important concept in NP-Complete theory. It can be mathematically represented as follows:

A £*P* B

It illustrates that problem B is as hard as problem A and solving problem A cannot be harder than problem B.

Karp reduction algorithm reduces a problem to another problem. It can be denoted as follows:

A £*P* B

The input of a problem is called instance. Mathematically, if there exists a polynomial time computable function f, such that for instance w,

"w, w Î *A* Û *f* (*w*) Î *B*

Then, the reduction is called Karp reduction.

So, the steps of Karp reduction can be said follows:

1. Construct f.
2. Show f is polynomial time computable.
3. Prove f is a reduction, i.e show for instance w,
1. If wÎA then f(w)ÎB
2. If f(w)ÎB then wÎA

Polynomial time Turing reduction is also called Cook Reduction. It must be observed that all Karp reductions are Cook Reductions but vice versa is not true and Turing Reduction and Oracle Reduction are synonymous.

Karp reduction is suitable for NP-C proof, Cook reduction is general and Karp reduction is suitable for decision problems. Cook reduction is applicable for all problems in general [2,3]

Example of Reduction

Consider the example of reducing Hamiltonian path problem to Hamiltonian cycle problem.

Consider the following graph based on [1] shown in Fig. 1.

In Hamiltonian cycle, the aim is to find Hamiltonian cycle. In Fig 1, the Hamiltonian cycle is given as 4-3-1-2-4. The aim of reduction is to reduce a Hamiltonian cycle to Hamiltonian path and vice versa. The Hamiltonian path problem can be stated as follows:

Instance: a directed graph G=(V,E) and two vertices s¹tÎV.

Hamiltonian Path Problem: To decide if there exists a path from s to t, which goes through each node once.

To illustrate reduction, let us first restate the problem as follows:

1. If there exists a Hamiltonian path ($v_0 = s, v_1, \ldots, v_n = t$) in the original graph, then
   ($u, v_0 = s, v_1, \ldots, v_n = t, u$) is a Hamiltonian cycle in the new graph. This is called completeness property.

2. (u,s) and (t,u) must be in any Hamiltonian cycle in the constructed graph, thus removing u yields a Hamiltonian path from s to t. This is called soundness property.

In other words, Hamiltonian path is converted to a cycle by adding a source vertex to form a cycle and by removing it the cycle to a path.

So, the proof can be shown as follows:

1. Construct f.
2. Show f is polynomial time computable.
3. Prove f is a reduction, i.e show:
1. If wÎHAMPATH then f(w)ÎHAMCYCLE
2. If f(w)ÎHAMCYCLE then wÎHAMPATH

NP-Complete proof outline

Using the concept of reduction, the NP-Complete proof can be done. The proof outline is given

as follows:

1. Take one existing well known NP-Complete Problem
2. Reduce the NP-Complete problem to the given problem to be proved.
3. Argue that the given problem is as hard as the well-known NP-Complete problem.

Proof of SAT is NP- Complete

Let us consider a simple case of proving Formula Satisfiability problem as NP-Complete. Let us

use the outline of NP-Complete proof outline.

1. Let us take a well-known NP-Complete problem called Circuit Satisfiability problem. It can be recollected from module 33, that Circuit Satisfiability problem is given as follows:

Given a Boolean circuit, consisting of gates such as, NOT, OR, AND, is there any set of inputs that makes the circuit output TRUE. Simultaneously, one can check for circuit output NO also. Circuit satisfiability is a NP-Complete problem as per Cook-Levin theorem.

2. Now take the problem of Formula Satisfiability problem, SAT, for which the NP-Complete proof is required. The formula Satisfiability problem is given as follows:

Given a Boolean formula, determine whether this formula is satisfiable or not. SAT problem formula consists of following components:

A literal : x1 or x1

A clause C : x1 v x2 v x3

A formula : conjunctive normal form

C1& C2 & … & Cm

So, the proof outline based on [1,2,3] is given as follows:

1. Circuzt-SAT is NP-Complete. Given an assignment, we can just check that each clause is covered in polynomial time. It is also possible to reduce a circuit for a formula in a polynomial time. Let us consider the circuit shown in Fig. 3.

3. In the third step, one can conclude that as any Circuit-SAT solution will satisfy the formula SAT instance and a *Circuit*-SAT solution can set variables giving a SAT solution, the problems are equivalent. Therefore, one can conclude that formula SAT is NP-Complete.

3-SAT is NP-Complete

One can extend the above proof for showing that 3-SAT is also NP-Complete. In 3-SAT or 3SAT, there must be exactly 3 literals per clause. The problem can be formulated as follows:

Given 3-CNF formula, Is there an assignment that makes the formula to evaluate to TRUE.

3-SAT is NP. Given an assignment, we can just check that each clause is covered. Based on the outline, one can say 3-SAT is hard.

To give a proof of 3-SAT, a well-known NP-Complete problem SAT can be taken. SAT can be converted to 3-SAT in a polynomial time as shown below:

We will transform each clause independently based on its *length*. Suppose a clause contains *k* literals:

1. if $k = 1$ (meaning $Ci = \{z1\}$), we can add in two new variables $v1$ and $v2$, and transform this into 4 clauses:

$\{v1, v2, z1\}$ $\{v1, Øv2, z1\}$ $\{Øv1, v2, z1\}$ $\{Øv1, Øv2, z1\}$

2. if $k = 2$ ($Ci = \{z1, z2\}$), we can add in one variable $v1$ and 2 new clauses: $\{v1, z1, z2\}$ $\{Øv1, z1, z2\}$

3. if $k = 3$ ($Ci = \{z1, z2, z3\}$), we move this clause as- is

4. if $k > 3$ ($Ci = \{z1, z2, \ldots, zk\}$) we can add in $k - 3$ new variables ($v1, \ldots, vk\text{-}3$) and $k - 2$ clauses:

$\{z1, z2, v1\}$ $\{Øv1, z3, v2\}$ $\{Øv2, z4, v3\}$ … $\{Øvk\text{-}3, zk\text{-}1, zk\}$

To prove 3-SAT is hard, a reduction from SAT to 3-SAT must be provided. This is done using the above said rules.

Then in third step, the NPC of 3-SAT can be argued like this: Since any SAT solution will satisfy the *3*-SAT instance and a *3*-SAT solution can set variables giving a SAT solution, the problems are equivalent. In other words, If there were *n* clauses and *m* distinct literals in the SAT instance, this transform takes $O(nm)$ time. Therefore, SAT = *3*-SAT.

Summary

One can conclude from this module 34 that

- Karp reduction reduces a problem A to problem B
- NP completeness proof is to reduce a given problem to a well known NP-Complete problem
- Using SAT, proofs can be given for many problems

Recent Advancements in Design and Analysis of Algorithms

The design and analysis of algorithms is a central area of computer science, focusing on developing efficient methods to solve problems and understanding their computational complexity. In recent years, there have been significant advancements across various aspects of algorithm theory, including improvements in algorithmic paradigms, the integration of

new mathematical techniques, and the practical application of these techniques in emerging fields like machine learning, quantum computing, and big data.

This detailed exploration covers the latest developments in the design and analysis of algorithms, highlighting key breakthroughs and emerging trends.

---

1. Advancements in Algorithmic Paradigms

a. Approximation Algorithms

Approximation algorithms are critical when exact solutions are computationally infeasible, particularly for NP-hard problems. Recent advancements include:

- **Improved Approximation Ratios**: Research has led to new approximation algorithms that provide better approximation ratios for many NP-hard problems, such as the **Traveling Salesman Problem (TSP)**, **Knapsack Problem**, and **Vertex Cover**. Some of the latest work involves refining the approximation factor for classical problems and extending it to more complex settings, like **online** or **dynamic settings**.
- **Parameterized Approximation**: There has been an increase in developing approximation algorithms that leverage the structure of the input, such as **fixed-parameter tractable (FPT)** algorithms. These algorithms allow for efficient approximation under certain problem parameters, providing much better practical performance than general-purpose approximation techniques.

b. Randomized Algorithms

Randomized algorithms use randomness as a tool for improving performance, especially for large-scale or complex problems. Recent developments include:

- **Derandomization**: A major trend has been to reduce or eliminate randomness in algorithms while maintaining efficiency. Advances in **derandomization techniques** have enabled deterministic algorithms to achieve similar performance to their randomized counterparts, especially in graph algorithms and network design problems.
- **Randomized Sublinear Algorithms**: A significant development is in the area of **sublinear algorithms** that operate on large datasets by sampling the input space efficiently. These algorithms are capable of solving problems like **estimating frequency moments** and **graph properties** (e.g., connectivity, cuts) in nearly constant time.

c. Online Algorithms

Online algorithms deal with problems where input is given sequentially, and decisions must be made without knowing the entire input. Recent developments include:

- **Competitive Analysis Refinements**: New theoretical frameworks have been proposed for analyzing the competitiveness of online algorithms, particularly in resource allocation problems, **caching**, and **load balancing**. These refinements improve the guarantees provided by online algorithms.
- **Learning-Augmented Online Algorithms**: The integration of **machine learning** with online algorithms has become a promising area. By using prior data or adjusting policies based on feedback, these algorithms can outperform traditional static strategies.

d. Data-Dependent Algorithms

A growing focus in algorithm design is **data-dependent analysis**, which involves designing algorithms that adapt based on specific features of the input data. Recent advancements include:

- **Sublinear Algorithms with Data Dependence**: There are new methods to improve the efficiency of algorithms by exploiting the inherent structure or patterns in the data. For example, **locality-sensitive hashing (LSH)** and **sparse approximation** algorithms exploit data sparsity or other properties to significantly reduce the time complexity in high-dimensional spaces.
- **Adaptive Algorithms**: Algorithms that adapt their behavior dynamically as they process data are being developed, particularly for applications like **data streams**, where input is too large to process entirely at once.

---

2. Mathematical and Theoretical Advancements

a. Graph Algorithms

Graph theory has long been a central area for algorithmic innovation, and there are recent advancements in both exact and approximate solutions to graph problems:

- **Dynamic Graph Algorithms**: As large networks evolve over time (e.g., social networks), there has been a push for **dynamic graph algorithms** that can update solutions incrementally as the graph changes, rather than recomputing everything from scratch. This includes advances in **dynamic shortest paths** and **dynamic connectivity** algorithms.
- **Spectral Graph Theory**: New insights into the spectral properties of graphs are enabling more efficient algorithms for problems like graph partitioning, clustering, and network flow. Spectral methods are also being applied in large-scale data analysis, such as **community detection** in social networks.
- **Sublinear Graph Algorithms**: Significant progress has been made in designing sublinear-time algorithms for graph problems. These algorithms allow for approximate solutions to

problems like **graph coloring**, **network centrality**, and **maximum flow** in massive graphs with reduced memory usage.

b. Mathematical Programming

Mathematical programming continues to be a fundamental technique for optimization problems. Recent innovations include:

- **Semi-Definite Programming (SDP)**: SDP has become central to many modern algorithms, particularly for **approximation algorithms** in NP-hard problems. New algorithms, particularly for **SDP relaxations**, have improved approximation guarantees for problems such as **Max-Cut**, **Max-2-SAT**, and other combinatorial optimization problems.
- **Convex Optimization**: Advances in **convex optimization** have provided better algorithms for large-scale machine learning problems. Algorithms that scale with **distributed optimization** techniques have become increasingly important for parallel and cloud-based systems.

---

3. Machine Learning and Algorithms

Machine learning (ML) and algorithms are increasingly intertwined. The latest advancements in the design and analysis of algorithms are heavily influenced by machine learning principles:

a. Learning-Based Algorithms

The integration of **supervised learning** and **reinforcement learning** has led to new classes of algorithms that adapt over time to optimize performance:

- **Reinforcement Learning for Algorithm Design**: RL is being used to learn algorithmic strategies for tasks like **sorting**, **searching**, and **optimization**. RL-based algorithms can improve over time, making them particularly effective for dynamic or real-time environments.
- **Neural Network-Driven Algorithms**: Algorithms that leverage deep learning techniques are being developed to approximate solutions to intractable problems, such as **integer programming**, **combinatorial optimization**, and **graph traversal**. These approaches, called **neural combinatorial optimization**, aim to train neural networks to learn good heuristics for complex combinatorial tasks.

b. Automated Algorithm Design

There has been a focus on **automating the design** of algorithms using machine learning. This includes:

- **Neural Architecture Search (NAS)**: NAS has been applied to optimize algorithmic architectures, such as designing more efficient neural network architectures or searching for optimal hyperparameters in machine learning algorithms.
- **Meta-Learning Algorithms**: Meta-learning, or "learning to learn," focuses on developing algorithms that can adapt to new tasks quickly. These algorithms are crucial for developing adaptable, general-purpose AI systems.

c. Federated Learning and Distributed Algorithms

In the context of **distributed machine learning**, federated learning has emerged as a new paradigm, where models are trained across decentralized devices (such as smartphones) without centralizing the data. Recent developments have focused on:

- **Optimized Federated Learning Algorithms**: New algorithms are being developed to improve the efficiency and communication overhead of federated learning systems, such as methods for adaptive model aggregation and **privacy-preserving algorithms** that protect sensitive user data.

---

4. Quantum Algorithms

Quantum computing is a rapidly growing field, and it has profound implications for algorithm design. Recent advances include:

a. Quantum Speedup

Many classical algorithms have quantum counterparts that promise significant speedup. Key recent advancements include:

- **Shor's Algorithm**: Shor’s algorithm for factoring large numbers has been further optimized to handle larger inputs and is a foundational result in quantum algorithm design.
- **Quantum Machine Learning**: Quantum algorithms are being developed for problems in machine learning, such as **quantum support vector machines (SVMs)** and **quantum clustering** algorithms. These methods promise exponential speedups for certain tasks, especially in high-dimensional spaces.

b. Quantum Approximation Algorithms

Quantum approximation algorithms are being refined to solve NP-hard problems. These involve using quantum states to represent problem instances and quantum operations to approximate the optimal solution.

---

5. Complexity Theory and Hardness Results

Recent developments in **computational complexity theory** include:

- **Fine-Grained Complexity**: Researchers have made progress in fine-grained complexity, which involves proving the hardness of problems at a finer level of granularity, such as understanding the exact time complexity of problems like **3-SAT** and **Shortest Paths**.
- **Average-Case Complexity**: New techniques in analyzing average-case hardness provide deeper insights into the practical difficulty of problems, leading to better algorithmic strategies.
- **Quantum Complexity Classes**: The classification of problems in quantum settings (e.g., **QMA**, **BQP**) has seen progress, helping identify which problems quantum computers can potentially solve faster than classical ones.