Advanced Data Structures and Algorithms

Classroom Assignment – Solutions to Algorithm and Complexity Theory Problems

Name: Nilesh Mukherjee
Roll No.: A125011
Programme: M.Tech, Computer Science and Engineering
Institution: International Institute of Information Technology, Bhubaneswar
Academic Year: 2025–2026

Question 1

Question: Prove that the time complexity of the recursive Heapify operation is $O(\log n)$ using the recurrence relation:

$$T(n) = T\left(\frac{2n}{3}\right) + O(1)$$

Solution

Given the recurrence relation:

$$T(n) = T\left(\frac{2n}{3}\right) + O(1)$$

At each recursive call, the problem size is reduced to $\frac{2n}{3}$ and a constant amount of work is performed.

Expanding the recurrence,

$$\begin{aligned} T(n) &= T\left(\frac{2n}{3}\right) + c \\ &= T\left(\left(\frac{2}{3}\right)^2 n\right) + c + c \\ &= T\left(\left(\frac{2}{3}\right)^3 n\right) + 3c \\ &\vdots \\ &= T\left(\left(\frac{2}{3}\right)^k n\right) + kc \end{aligned}$$

The recursion terminates when the input size becomes constant:

$$\left(\frac{2}{3}\right)^k n = 1$$

Taking logarithms on both sides,

$$k \log\left(\frac{2}{3}\right) + \log n = 0$$

$$k = \frac{\log n}{\log\left(\frac{3}{2}\right)}$$

Since $\log\left(\frac{3}{2}\right)$ is a positive constant, we get:

$$k = O(\log n)$$

Substituting this value into the expanded recurrence,

$$T(n) = T(1) + O(\log n)$$

As $T(1) = O(1)$, the overall time complexity is:

$$T(n) = O(\log n)$$

Hence, the time complexity of the recursive Heapify operation is:

$$\boxed{O(\log n)}$$

Question 2

Question: In an array of size n representing a binary heap, prove that all leaf nodes are located at indices from $\left\lfloor \frac{n}{2} \right\rfloor + 1$ to n.

Solution

Consider a binary heap stored in an array of size n using 1-based indexing.

In a binary heap, for any node located at index i:

$$\text{Left child index} = 2i$$

$$\text{Right child index} = 2i + 1$$

A node is a leaf node if it has no children. Therefore, for a node at index i to be a leaf node, both of the following conditions must hold:

$$2i > n \quad \text{and} \quad 2i + 1 > n$$

From the first condition,

$$2i > n$$

Dividing both sides by 2,

$$i > \frac{n}{2}$$

Since i must be an integer index, this implies:

$$i \geq \left\lfloor \frac{n}{2} \right\rfloor + 1$$

The maximum possible index in the array is n. Hence, all leaf nodes are located at indices:

$$\left\lfloor \frac{n}{2} \right\rfloor + 1 \leq i \leq n$$

Therefore, all leaf nodes in a binary heap of size n are located at indices from $\left\lfloor \frac{n}{2} \right\rfloor + 1$ to n.

$$\boxed{\left\lfloor \frac{n}{2} \right\rfloor + 1 \text{ to } n}$$

Question 3

Question:

(a) Show that in any heap containing n elements, the number of nodes at height h is at most:

$$\left\lceil \frac{n}{2^{h+1}} \right\rceil$$

(b) Using the above result, prove that the time complexity of the Build-Heap algorithm is $O(n)$.

Solution

(a) Consider a binary heap containing n elements. The height of a node is defined as the number of edges on the longest downward path from that node to a leaf.

In a binary heap, nodes at height h are internal nodes whose subtrees have height h. Each such node must have at least 2^h nodes in its subtree. Therefore, if there are x nodes at height h, the total number of nodes in the heap must satisfy:

$$x \cdot 2^h \leq n$$

Dividing both sides by 2^h,

$$x \leq \frac{n}{2^h}$$

Since only internal nodes can have height $h \geq 0$, the maximum number of such nodes is bounded by:

$$x \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$$

Hence, in a heap containing n elements, the number of nodes at height h is at most:

$$\left\lceil \frac{n}{2^{h+1}} \right\rceil$$

(b) The Build-Heap algorithm works by calling Heapify on all non-leaf nodes, starting from the lowest internal node up to the root.

The time required to Heapify a node at height h is $O(h)$, since the node may move down at most h levels.

From part (a), the number of nodes at height h is at most:

$$\left\lceil \frac{n}{2^{h+1}} \right\rceil$$

Therefore, the total time complexity of Build-Heap is given by:

$$T(n) = \sum_{h=0}^{\lfloor \log n \rfloor} \left(\frac{n}{2^{h+1}} \cdot O(h) \right)$$

$$T(n) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right)$$

The series $\sum_{h=0}^{\infty} \frac{h}{2^h}$ converges to a constant.

Hence,

$$T(n) = O(n)$$

Therefore, the time complexity of the Build-Heap algorithm is:

$$\boxed{O(n)}$$

Question 4

Question: Explain the LU decomposition of a matrix using Gaussian Elimination. Clearly describe each step involved in the process.

Solution

LU decomposition is a fundamental matrix factorization technique used in numerical linear algebra. The main idea is to decompose a given square matrix A into the product of two triangular matrices:

$$A = LU$$

where L is a lower triangular matrix with unit diagonal elements and U is an upper triangular matrix. This decomposition is particularly useful for solving systems of linear equations, computing determinants, and finding matrix inverses.

Solving a system of equations $Ax = b$ directly using Gaussian elimination requires $O(n^3)$ time. If the same matrix A is used with multiple right-hand side vectors b, repeating Gaussian elimination becomes computationally expensive. LU decomposition overcomes this issue by separating the elimination phase from the solving phase. Once the matrix A is decomposed into L and U, the system can be solved efficiently by first solving $Ly = b$ using forward substitution and then solving $Ux = y$ using backward substitution. Each substitution step takes only $O(n^2)$ time.

Gaussian elimination works by transforming a matrix into an upper triangular form by eliminating elements below the main diagonal using elementary row operations. These elimination steps naturally give rise to the matrices L and U.

Consider a square matrix:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

In the first step, the element a_{11} is used as the pivot to eliminate the elements below it. The multipliers are computed as:

$$l_{21} = \frac{a_{21}}{a_{11}}, \quad l_{31} = \frac{a_{31}}{a_{11}}$$

and the corresponding row operations are applied:

$$R_2 \leftarrow R_2 - l_{21}R_1, \quad R_3 \leftarrow R_3 - l_{31}R_1$$

After elimination, the matrix takes the form:

$$U^{(1)} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & u_{22} & u_{23} \\ 0 & u_{32} & u_{33} \end{bmatrix}$$

The multipliers used in the elimination process are stored in the lower triangular matrix:

$$L = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & 0 & 1 \end{bmatrix}$$

In the second step, the element u_{22} is used as the pivot to eliminate u_{32}. The multiplier

is:

$$l_{32} = \frac{u_{32}}{u_{22}}$$

and the row operation performed is:

$$R_3 \leftarrow R_3 - l_{32}R_2$$

This results in the final upper triangular matrix:

$$U = \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

The lower triangular matrix is updated as:

$$L = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix}$$

Thus, the original matrix can be expressed as:

$$A = LU$$

LU decomposition exists provided all pivot elements are non-zero. In practice, pivoting may be required to ensure numerical stability, though for positive definite matrices, pivoting is not necessary. This decomposition is widely used in numerical simulations, scientific computing, and efficient linear system solvers.

Question 5

Question: Solve the following recurrence relation arising from the LUP decomposition solve procedure:

$$T(n) = \sum_{i=1}^{n} \left[O(1) + \sum_{j=1}^{i-1} O(1) \right] + \sum_{i=1}^{n} \left[O(1) + \sum_{j=i+1}^{n} O(1) \right]$$

Solution

The given recurrence relation consists of two summation terms. We simplify each term separately.

Consider the first summation:

$$\sum_{i=1}^{n} \left[O(1) + \sum_{j=1}^{i-1} O(1) \right]$$

The inner summation $\sum_{j=1}^{i-1} O(1)$ runs $(i-1)$ times, hence:

$$\sum_{j=1}^{i-1} O(1) = O(i)$$

Therefore, the first summation becomes:

$$\sum_{i=1}^{n} O(i)$$

Now consider the second summation:

$$\sum_{i=1}^{n} \left[O(1) + \sum_{j=i+1}^{n} O(1) \right]$$

The inner summation $\sum_{j=i+1}^{n} O(1)$ runs $(n-i)$ times, hence:

$$\sum_{j=i+1}^{n} O(1) = O(n-i)$$

Thus, the second summation becomes:

$$\sum_{i=1}^{n} O(n-i)$$

Now we combine both results:

$$T(n) = \sum_{i=1}^{n} O(i) + \sum_{i=1}^{n} O(n-i)$$

Using standard summation results:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$$

$$\sum_{i=1}^{n} (n-i) = \sum_{k=0}^{n-1} k = \frac{n(n-1)}{2} = O(n^2)$$

Hence,

$$T(n) = O(n^2) + O(n^2)$$

$$T(n) = O(n^2)$$

Therefore, the time complexity of the given recurrence relation is:

$$\boxed{O(n^2)}$$

Question 6

Question: Prove that if matrix A is non-singular, then its Schur complement is also non-singular.

Solution

The Schur complement is an important concept in matrix analysis and numerical linear algebra. It frequently appears in LU decomposition, block matrix factorization, and in solving systems of linear equations. We are required to prove that if a matrix A is non-singular, then its Schur complement is also non-singular.

Let the matrix A be partitioned into block form as:

$$A = \begin{bmatrix} B & C \\ D & E \end{bmatrix}$$

where B is a square and invertible matrix, C and D are rectangular matrices, and E is a square matrix.

The Schur complement of the block B in matrix A is defined as:

$$S = E - DB^{-1}C$$

Intuitively, the Schur complement represents the effect of eliminating block B from the system.

Since B is invertible, matrix A can be factorized as:

$$A = \begin{bmatrix} I & 0 \\ DB^{-1} & I \end{bmatrix} \begin{bmatrix} B & C \\ 0 & E - DB^{-1}C \end{bmatrix}$$

The first matrix is a lower triangular block matrix with identity matrices on the diagonal, and the second matrix is an upper triangular block matrix. Their product reconstructs the original matrix A.

Taking determinants on both sides, we obtain:

$$\det(A) = \det \begin{bmatrix} I & 0 \\ DB^{-1} & I \end{bmatrix} \cdot \det \begin{bmatrix} B & C \\ 0 & S \end{bmatrix}$$

The determinant of the first matrix is 1, since it is triangular with identity matrices on the diagonal. The determinant of the second matrix is the product of the determinants of its diagonal blocks. Hence,

$$\det(A) = \det(B) \cdot \det(S)$$

Given that A is non-singular, $\det(A) \neq 0$. Since B is invertible, $\det(B) \neq 0$. Therefore, it must be that:

$$\det(S) \neq 0$$

This implies that the Schur complement S is non-singular.

Hence, if matrix A is non-singular, then its Schur complement is also non-singular.

Schur complement is non-singular

Question 7

Question: Prove that positive-definite matrices are suitable for LU decomposition and do not require pivoting to avoid division by zero in the recursive strategy.

Solution

In numerical linear algebra, LU decomposition may fail if a pivot element becomes zero, leading to division by zero during Gaussian elimination. Pivoting is commonly introduced to avoid this issue. However, for an important class of matrices known as positive-definite matrices, LU decomposition can be safely performed without pivoting.

A real symmetric matrix $A \in \mathbb{R}^{n \times n}$ is said to be positive definite if it satisfies:

$$x^T Ax > 0 \quad \text{for all } x \neq 0$$

Positive-definite matrices possess several important properties. All eigenvalues of A are strictly positive, all leading principal minors are positive, and consequently, A is non-singular. These properties play a crucial role in LU decomposition.

In LU decomposition using Gaussian elimination, pivot elements are the diagonal entries of the upper triangular matrix U. These pivot elements appear in the denominator while computing the elimination multipliers. If any pivot becomes zero, the algorithm

breaks down, which is why pivoting is usually required for general matrices.

For a positive-definite matrix A, all leading principal minors satisfy:

$$\det(A_k) > 0 \quad \text{for } k = 1, 2, \ldots, n$$

where A_k denotes the $k \times k$ leading principal submatrix. Since pivot elements correspond to these leading principal pivots generated during Gaussian elimination, none of the pivots can be zero. Therefore, division by zero never occurs during the elimination process.

In the recursive LU decomposition strategy, the matrix is partitioned into blocks and Schur complements are computed at each step. Positive-definiteness is preserved under Schur complementation. Hence, at every recursive level, the resulting submatrices remain positive definite and continue to have non-zero pivot elements. This guarantees that the recursion proceeds safely without pivoting.

As a result, LU decomposition of a positive-definite matrix can be carried out without row exchanges, and the factorization:

$$A = LU$$

exists with strictly positive diagonal entries in U. In fact, positive-definite matrices admit an even stronger factorization known as the Cholesky decomposition.

Hence, positive-definite matrices are well-suited for LU decomposition and do not require pivoting to avoid division by zero in the recursive strategy.

Positive-definite matrices allow LU decomposition without pivoting

Question 8

Question: For finding an augmenting path in a graph, should Breadth First Search (BFS) or Depth First Search (DFS) be applied? Justify your answer.

Solution

An augmenting path is a path from the source to the sink in a residual graph along which additional flow can be pushed. The choice of search strategy for finding such a path plays a crucial role in the correctness and efficiency of maximum flow algorithms.

Both Breadth First Search (BFS) and Depth First Search (DFS) can be used to find an augmenting path. However, Breadth First Search is preferred in practice, especially in the Edmonds–Karp algorithm.

Using BFS ensures that the augmenting path found is the shortest path in terms of the number of edges. This property is important because it guarantees that the length

of the shortest augmenting path never decreases during the algorithm. As a result, each edge can become critical only a limited number of times, which leads to a polynomial time bound.

In contrast, DFS may find arbitrarily long augmenting paths. While it can still lead to a correct solution, it does not provide any guarantee on the number of iterations required. In the worst case, using DFS may result in exponential time complexity due to repeatedly choosing inefficient augmenting paths.

The Edmonds–Karp algorithm, which is a specific implementation of the Ford–Fulkerson method, explicitly uses BFS to find augmenting paths. By always selecting the shortest augmenting path, the algorithm achieves a time complexity of $O(VE^2)$, where V is the number of vertices and E is the number of edges.

Therefore, Breadth First Search should be applied for finding augmenting paths, as it ensures better performance, avoids pathological cases, and guarantees polynomial-time termination.

BFS is preferred over DFS for finding augmenting paths

Question 9

Question: Explain why Dijkstra's algorithm cannot be applied to graphs with negative edge weights.

Solution

Dijkstra's algorithm is a greedy algorithm used to find the shortest paths from a source vertex to all other vertices in a graph with non-negative edge weights. The correctness of the algorithm relies on the assumption that once a vertex is selected with the minimum tentative distance, its shortest path distance is final and cannot be improved.

This assumption fails in the presence of negative edge weights. When a graph contains negative edges, a vertex that has already been finalized by Dijkstra's algorithm may later receive a shorter path through another vertex using a negative-weight edge. Since Dijkstra's algorithm never revisits finalized vertices, it cannot correct this error.

As a result, the algorithm may produce incorrect shortest path distances. The greedy choice made at each step becomes invalid because negative edges violate the monotonicity property required for correctness.

For example, consider a graph where a vertex appears to have the shortest distance initially, but a different path reaching it later through a negative edge yields a smaller total distance. Dijkstra's algorithm would fail to detect this improvement.

Therefore, Dijkstra's algorithm cannot be applied to graphs with negative edge weights. For such graphs, algorithms like Bellman–Ford are used instead, as they are capable of handling negative edge weights and detecting negative cycles.

Dijkstra's algorithm fails in the presence of negative edge weights

Question 10

Question: Prove that every connected component of the symmetric difference of two matchings in a graph G is either a path or an even-length cycle.

Solution

Matchings are a fundamental concept in graph theory and play an important role in network flow, bipartite matching, and combinatorial optimization. Let $G = (V, E)$ be an undirected graph, and let M_1 and M_2 be two matchings in G.

The symmetric difference of M_1 and M_2 is defined as:

$$M_1 \oplus M_2 = (M_1 \setminus M_2) \cup (M_2 \setminus M_1)$$

That is, $M_1 \oplus M_2$ consists of edges that belong to exactly one of the two matchings.

Since M_1 and M_2 are matchings, no vertex in G is incident to more than one edge from M_1, and similarly no vertex is incident to more than one edge from M_2. Therefore, in the subgraph formed by $M_1 \oplus M_2$, each vertex can be incident to at most two edges: at most one from M_1 and at most one from M_2. Hence, every vertex in $M_1 \oplus M_2$ has degree at most two.

This degree restriction strongly limits the possible structure of connected components. A graph in which every vertex has degree at most two can only consist of paths and cycles. If a connected component contains a vertex of degree one, it must be a path. If all vertices in a component have degree two, the component must be a cycle.

Furthermore, edges in $M_1 \oplus M_2$ must alternate between M_1 and M_2 along any connected component, since adjacent edges cannot belong to the same matching. In the case of a cycle, this alternating structure implies that the cycle must contain an even number of edges. An odd-length cycle would force two adjacent edges to belong to the same matching, which is not possible.

Therefore, every connected component of the symmetric difference $M_1 \oplus M_2$ is either a path or an even-length cycle.

Each connected component of $M_1 \oplus M_2$ is a path or an even-length cycle

Question 11

Question: Define the class Co-NP. Explain the type of problems that belong to this complexity class.

Solution

In computational complexity theory, decision problems are classified based on how efficiently their solutions can be verified. One such important class is Co-NP.

A decision problem belongs to the class NP if a "YES" answer can be verified in polynomial time given a suitable certificate. In contrast, the class Co-NP focuses on problems whose "NO" answers can be verified efficiently.

Formally, a language L belongs to the class Co-NP if and only if its complement $\overline{L}$ belongs to NP:

$$L \in \text{Co-NP} \iff \overline{L} \in \text{NP}$$

Equivalently, a problem is in Co-NP if there exists a polynomial-time verifiable certificate for every "NO" instance of the problem.

Intuitively, NP contains problems where proving that a solution exists is easy, whereas Co-NP contains problems where proving that no solution exists is easy. Thus, Co-NP captures problems involving efficient verification of incorrectness or non-existence.

A classic example of a Co-NP problem is **UNSAT**, which asks whether a Boolean formula is unsatisfiable. A proof that no satisfying assignment exists serves as a certificate for the "NO" instance. Another example is the **TAUTOLOGY** problem, which asks whether a Boolean formula evaluates to true for all possible assignments. Its complement belongs to NP, as a counterexample assignment can be verified efficiently. Similarly, determining whether a number is composite also belongs to Co-NP, since a factorization acts as a polynomial-time verifiable certificate.

The relationship between NP and Co-NP is a major open problem in complexity theory. It is known that:

$$P \subseteq NP \cap \text{Co-NP}$$

but it is not known whether $NP = \text{Co-NP}$.

Hence, Co-NP is the class of decision problems for which "NO" instances admit efficiently verifiable certificates.

Co-NP consists of problems whose NO answers can be verified in polynomial time

Question 12

Question: Given a Boolean circuit instance whose output evaluates to true, explain how the correctness of the result can be verified in polynomial time using Depth First Search (DFS).

Solution

Boolean circuits are widely used in complexity theory to represent computations. A Boolean circuit consists of input variables, logic gates such as AND, OR, and NOT, and a designated output gate. Such a circuit can be represented as a directed acyclic graph (DAG), where nodes correspond to gates or inputs and edges represent the flow of computation.

Suppose we are given a Boolean circuit along with an assignment to its input variables, and the output of the circuit evaluates to TRUE. The task is to verify whether this output is correct for the given assignment.

This verification can be performed efficiently using Depth First Search (DFS). Starting from the output gate, DFS is applied recursively to traverse the circuit backward toward the input gates. During this traversal, the correctness of each gate is verified based on the values of its input gates. For an AND gate, the output is TRUE only if all its inputs are TRUE. For an OR gate, the output is TRUE if at least one input is TRUE. For a NOT gate, the output is the negation of its input.

The DFS traversal ensures that each gate is visited only once and that the verification proceeds in a bottom-up manner. Since the circuit is acyclic, there is no repeated work, and the value of each gate is computed and checked exactly once.

The total time required for this verification is proportional to the number of gates and connections in the circuit. Thus, the time complexity is:

$$O(V + E)$$

where V is the number of gates and E is the number of edges. Since the size of the circuit is polynomial in the input size, the verification runs in polynomial time.

Therefore, given a Boolean circuit whose output evaluates to TRUE, the correctness of the result can be verified efficiently using DFS. This demonstrates why the Boolean Circuit Value Problem belongs to the complexity class NP.

Boolean circuit correctness can be verified in polynomial time using DFS

Question 13

Question: Is the 3-SAT (3-CNF-SAT) problem NP-Hard? Justify your answer.

Solution

The 3-SAT problem is a special case of the Boolean satisfiability problem. In 3-SAT, a Boolean formula is written in conjunctive normal form (CNF), where:

- Each clause contains exactly three literals
- Each literal is a variable or its negation

The problem asks whether there exists an assignment of truth values to variables that makes the entire formula true.

To understand whether 3-SAT is NP-Hard, it is useful to recall the meaning of NP-Hardness:

- A problem is NP-Hard if every problem in NP can be reduced to it in polynomial time
- Intuitively, this means the problem is at least as hard as all problems whose solutions can be verified efficiently

The general SAT problem was the first problem proven to be NP-Complete, as established by Cook's Theorem. This result showed that SAT captures the full computational difficulty of the class NP.

The key conceptual insight behind 3-SAT is that restricting clauses to exactly three literals does not significantly reduce the expressive power of the problem. In particular:

- Any Boolean formula in general CNF form can be transformed into an equivalent 3-CNF formula
- This transformation can be done in polynomial time
- Additional variables may be introduced, but satisfiability is preserved

As a result, the original formula is satisfiable if and only if the resulting 3-CNF formula is satisfiable. This means that solving 3-SAT would allow us to solve SAT itself.

Since:

- SAT is NP-Complete
- SAT can be reduced to 3-SAT in polynomial time

it follows that 3-SAT is NP-Hard. Furthermore, given a proposed satisfying assignment, the correctness of a 3-SAT solution can be verified in polynomial time, which places 3-SAT in NP as well.

The importance of 3-SAT lies in the fact that computational hardness does not require complex clause structures. Even with a very simple and uniform form—clauses of only three literals—the problem still captures the full difficulty of NP.

Hence, 3-SAT is NP-Hard, and in fact, it is one of the most fundamental NP-Complete problems in computational complexity theory.

Yes, 3-SAT is NP-Hard (and in fact NP-Complete)

Question 14

Question: Is the 2-SAT problem NP-Hard? Can it be solved in polynomial time? Explain your reasoning.

Solution

The 2-SAT problem is a restricted form of the Boolean satisfiability problem in which each clause contains at most two literals. Although it may appear similar to 3-SAT, the computational behavior of 2-SAT is fundamentally different.

To reason about its complexity, it is important to understand what makes problems NP-Hard. NP-Hard problems are expressive enough to encode arbitrary logical constraints, making them computationally difficult. The key insight is that 2-SAT lacks this expressive power due to its strong structural restrictions.

The crucial idea behind solving 2-SAT efficiently is the **implication graph**. Each clause of the form $(a \vee b)$ can be rewritten as logical implications:

$$(\neg a \Rightarrow b) \quad \text{and} \quad (\neg b \Rightarrow a)$$

Using this transformation:

- Each literal becomes a vertex in a directed graph
- Each implication becomes a directed edge
- The entire formula is represented as an implication graph

The satisfiability condition for 2-SAT is simple and structural:

- If a variable and its negation belong to the same strongly connected component (SCC), the formula is unsatisfiable

- Otherwise, the formula is satisfiable

Strongly connected components can be computed efficiently using graph algorithms such as DFS-based Kosaraju's or Tarjan's algorithm. These algorithms run in linear time with respect to the number of vertices and edges in the graph.

As a result:

- 2-SAT can be solved in polynomial time (in fact, linear time)
- No exponential search is required

This explains why 2-SAT is **not NP-Hard** (unless $P = NP$). The structural limitation of having only two literals per clause makes the problem tractable. In contrast, allowing three literals per clause (as in 3-SAT) destroys this structure and leads to NP-Hardness.

2-SAT represents a boundary case in satisfiability problems:

- 2-SAT: restricted structure $\Rightarrow$ efficient algorithms
- 3-SAT: greater expressive power $\Rightarrow$ computational hardness

Thus, the complexity difference between 2-SAT and 3-SAT highlights how small syntactic changes can cause major shifts in computational difficulty.

2-SAT is not NP-Hard and can be solved in polynomial time