Advanced Data Structures and Algorithms

Classroom Assignments – Solutions with Explaination

Name: Rishab Biswas

Roll No: A125017

Programme: M.Tech (Computer Science and Engineering)

Institution: International Institute of Information Technology, Bhubaneswar

Academic Year: 2025–2026

Contents

1 Time Complexity of Recursive Heapify **2**

2 Leaf Nodes in a Binary Heap **2**

3 Build-Heap Algorithm **3**

3.1 (a) Nodes at Height h 3

3.2 (b) Why Build-Heap is $O(n)$ 3

4 LU Decomposition using Gaussian Elimination **4**

5 Solving the Recurrence Relation in LUP Decomposition **9**

6 Non-Singularity of the Schur Complement **11**

7 Positive-Definite Matrices and LU Decomposition without Pivoting **14**

8 BFS vs DFS for Augmenting Paths **16**

9 Why Dijkstra Fails with Negative Weights **16**

10 Structure of the Symmetric Difference of Two Matchings **17**

11 Class Co-NP **19**

12 Verification of Boolean Circuit Output using DFS **21**

13 NP-Hardness of the 3-SAT Problem **22**

1 Time Complexity of Recursive Heapify

Understanding Heapify

Heapify is an operation used in a binary heap to restore the heap property after an element is misplaced. It works by comparing a node with its children and swapping it with the larger (or smaller) child, then repeating this process recursively.

Given Recurrence

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

This recurrence tells us:

- Each recursive call works on a smaller heap of size at most $\frac{2n}{3}$
- Each call performs only a constant number of comparisons and swaps

Key Idea

Every time Heapify is called, we move one level down the heap. Since a binary heap has height proportional to $\log n$, Heapify cannot recurse more than $\log n$ times.

Solving the Recurrence

After k recursive calls:

$$n\left(\frac{2}{3}\right)^k = 1 \Rightarrow k = \log_{3/2} n = O(\log n)$$

At each level, work done is constant.

Final Result

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

This confirms that Heapify is efficient even for large heaps.

■

2 Leaf Nodes in a Binary Heap

Heap Representation

A binary heap is stored in an array using level-order traversal.

For a node at index i:

- Parent index = $\lfloor i/2 \rfloor$
- Left child = $2i$
- Right child = $2i + 1$

When is a Node a Leaf?

A node is a leaf if it has no children. This means:

$$2i > n \Rightarrow i > \left\lfloor \frac{n}{2} \right\rfloor$$

Conclusion

All nodes from index:

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

are leaf nodes.

■

3 Build-Heap Algorithm

3.1 (a) Nodes at Height h

Understanding Height

- Height 0: Leaf nodes
- Height increases as we move upwards

At each higher level, the number of nodes roughly halves. Thus, nodes at height h are at most:

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

3.2 (b) Why Build-Heap is $O(n)$

Common Misconception

Many assume Build-Heap is $O(n \log n)$ because Heapify is $O(\log n)$. This is incorrect.

Correct Analysis

- Nodes near the bottom have small height (cheap to heapify)
- Only a few nodes are near the root (expensive heapify)

Total cost:

$$\sum_{h=0}^{\log n} \frac{n}{2^{h+1}} \cdot O(h)$$

This summation converges to:

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

■

4 LU Decomposition using Gaussian Elimination

Introduction

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
- U is an **upper triangular matrix**

This decomposition is especially useful for efficiently solving systems of linear equations of the form $Ax = b$, computing determinants, and finding matrix inverses.

Motivation for LU Decomposition

Solving a system $Ax = b$ directly using Gaussian elimination requires $O(n^3)$ time. If multiple right-hand sides b are given, repeating elimination becomes expensive.

LU decomposition separates the elimination step from the solving step:

- Decompose $A = LU$ once
- Solve $Ly = b$ using forward substitution
- Solve $Ux = y$ using backward substitution

Both substitutions take only $O(n^2)$ time.

Gaussian Elimination Overview

Gaussian elimination transforms a matrix into an upper triangular form by eliminating entries below the diagonal using row operations. These elimination steps are exactly what construct matrices L and U.

Step-by-Step LU Decomposition Process

Consider a square matrix:

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

Step 1: First Elimination Column

To eliminate elements below a_{11}, we compute multipliers:

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

Row operations:

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

The updated matrix becomes:

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

The multipliers are stored in matrix L:

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

Step 2: Second Elimination Column

Now eliminate u_{32} using pivot u_{22}:

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

Apply:

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

Final upper triangular matrix:

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

Update L:

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

Final LU Factorization

Thus, the original matrix satisfies:

$$A = LU$$

where:

- L contains all elimination multipliers
- U is the result of Gaussian elimination

Important Observations

- LU decomposition exists if all pivot elements are non-zero
- Pivoting may be required for numerical stability
- For positive definite matrices, pivoting is not required

Applications of LU Decomposition

- Solving linear systems efficiently
- Computing determinants:

$$\det(A) = \det(L)\det(U)$$

- Matrix inversion
- Numerical simulations and scientific computing

Conclusion

LU decomposition using Gaussian elimination systematically converts a matrix into triangular factors. It provides both theoretical insight and computational efficiency, making it a cornerstone technique in numerical algorithms.

■

Algorithm: LU Decomposition using Gaussian Elimination

The LU decomposition algorithm is derived directly from the steps of Gaussian elimination. During elimination, the multipliers used to eliminate lower triangular elements are stored in matrix L, while the transformed matrix becomes the upper triangular matrix U.

Input and Output

- **Input:** Square matrix $A \in \mathbb{R}^{n \times n}$
- **Output:** Lower triangular matrix L and upper triangular matrix U such that $A = LU$

Algorithm Steps

1. Initialize:
   - Set $L = I_n$ (identity matrix of order n)
   - Set $U = A$
2. For $k = 1$ to $n - 1$ **(pivot column)**:
   (a) For $i = k + 1$ to n **(rows below pivot)**:
      i. Compute the elimination multiplier:

$$L[i, k] = \frac{U[i, k]}{U[k, k]}$$

      ii. Eliminate the entry below the pivot:

$$U[i, j] = U[i, j] - L[i, k] \cdot U[k, j], \quad \forall j = k \text{ to } n$$

After completing all iterations, matrix U becomes upper triangular and matrix L contains all elimination multipliers below the diagonal.

$$\boxed{A = LU}$$

Explanation of Algorithm

- The outer loop selects the pivot element in each column
- The inner loops eliminate all entries below the pivot
- Multipliers used for elimination are stored in L
- The remaining matrix after elimination forms U

This process is mathematically equivalent to Gaussian elimination.

Time Complexity Analysis

We now compute the time complexity of LU decomposition using Gaussian elimination.

Operation Count

- The outer loop runs $(n-1)$ times
- For each k, the loop over i runs $(n-k)$ times
- For each (k, i) pair, the loop over j runs $(n-k+1)$ times

Hence, the total number of basic arithmetic operations is:

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

Simplification

$$= \sum_{k=1}^{n-1} (n-k)(n-k+1)$$

$$= O(n^3)$$

Final Time Complexity

$$\boxed{\text{Time Complexity of LU Decomposition} = O(n^3)}$$

Space Complexity

- Matrix L: $O(n^2)$
- Matrix U: $O(n^2)$

$$\boxed{\text{Space Complexity} = O(n^2)}$$

Remarks

- The algorithm fails if a pivot element becomes zero
- Pivoting leads to LUP decomposition for numerical stability
- For positive definite matrices, pivoting is not required

5 Solving the Recurrence Relation in LUP Decomposition

Problem Statement

The time complexity of the LUP decomposition solve procedure is given by the recurrence relation:

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

Our goal is to simplify this expression and determine the overall time complexity.

Understanding the Recurrence

This recurrence arises due to:

- Forward substitution while solving $Ly = b$
- Backward substitution while solving $Ux = y$

Each substitution involves nested loops that depend on matrix size n.

Analysis of the First Summation

Consider the first term:

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

Inner Summation

For a fixed i, the inner summation runs from $j = 1$ to $i - 1$. Thus, it executes $i - 1$ constant-time operations:

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

Outer Summation

Substituting into the outer summation:

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

This is an arithmetic series:

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

Hence, the first summation contributes:

$$O(n^2)$$

Analysis of the Second Summation

Now consider the second term:

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

Inner Summation

For a fixed i, the inner summation runs from $j = i + 1$ to n, which gives $n - i$ iterations:

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

Outer Summation

Substituting:

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

This is again an arithmetic series:

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

Hence, the second summation also contributes:

$$O(n^2)$$

Total Time Complexity

Combining both parts:

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

Final Result

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

Interpretation in LUP Decomposition

- The $O(n^2)$ complexity arises from solving triangular systems
- LU decomposition itself costs $O(n^3)$
- Once decomposed, each solve step is quadratic

Conclusion

The recurrence correctly captures the computational cost of forward and backward substitution steps in LUP decomposition, and its solution confirms that the solve phase runs in quadratic time.

■

6 Non-Singularity of the Schur Complement

Introduction

The Schur complement is an important concept in matrix analysis and numerical linear algebra. It frequently appears in LU decomposition, block matrix factorization, and solving linear systems.

This question asks us to prove that:

If a matrix A is non-singular, then its Schur complement is also non-singular.

Matrix Partitioning

Let the matrix A be partitioned into block form as follows:

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

where:

- B is a square and invertible matrix
- C and D are rectangular matrices
- E is a square matrix

Definition of Schur Complement

The Schur complement of 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.

Key Intuition Behind the Proof

The main idea of the proof is based on contradiction:

- Assume that A is non-singular
- Suppose that the Schur complement S is singular
- Show that this assumption leads to A being singular
- This contradicts the original assumption

Hence, S must be non-singular.

Formal Proof

Since B is invertible, we can factor matrix A as:

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

Explanation of the Factorization

- The first matrix is a lower triangular block matrix
- The second matrix is an upper triangular block matrix
- Their product reconstructs matrix A

Determinant Argument

Taking determinants on both sides:

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

Since:

- $\det(I) = 1$
- Determinant of a triangular matrix equals the product of diagonal blocks

We get:

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

Contradiction Argument

- Given: A is non-singular $\Rightarrow \det(A) \neq 0$
- B is invertible $\Rightarrow \det(B) \neq 0$

Therefore:

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

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

Conclusion

If matrix A is non-singular, then its Schur complement S is also non-singular

Why This Result Is Important

- Ensures correctness of block LU decomposition
- Guarantees stability in recursive matrix algorithms
- Widely used in numerical solvers and optimization

■

7 Positive-Definite Matrices and LU Decomposition without Pivoting

Introduction

In numerical linear algebra, LU decomposition may sometimes fail if a pivot element becomes zero or very small. To avoid this, pivoting is often introduced. However, for an important class of matrices called **positive-definite matrices**, LU decomposition can be safely performed without pivoting.

This question asks us to explain why:

> *Positive-definite matrices are suitable for LU decomposition and do not require pivoting to avoid division by zero in the recursive strategy.*

Definition of Positive-Definite Matrix

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

$$x^T A x > 0 \quad \forall x \neq 0$$

This definition ensures that the quadratic form associated with A is always strictly positive for any non-zero vector x.

Key Properties of Positive-Definite Matrices

Positive-definite matrices possess several important properties:

- All eigenvalues of A are strictly positive
- All leading principal minors of A are positive
- A is non-singular

These properties are crucial for understanding why LU decomposition works without pivoting.

Role of Pivot Elements in LU Decomposition

In LU decomposition using Gaussian elimination:

- Pivot elements are the diagonal elements of matrix U
- Each pivot element appears in the denominator while computing elimination multipliers

If any pivot element becomes zero, division by zero occurs and the algorithm fails. Pivoting is generally used to avoid this situation.

Why Pivoting Is Not Required for Positive-Definite Matrices

For a positive-definite matrix A:

- All leading principal minors are positive
- This implies that all pivots generated during Gaussian elimination are non-zero
- Therefore, no division by zero occurs

More formally, during LU decomposition:

$$\det(A_k) > 0 \quad \forall k = 1, 2, \ldots, n$$

where A_k denotes the $k \times k$ leading principal submatrix.

Since pivot elements correspond to ratios of determinants of these submatrices, they are guaranteed to be non-zero.

Connection to Recursive LU Strategy

In recursive LU decomposition:

- The matrix is split into blocks
- Schur complements are computed at each step

Positive-definiteness is preserved under Schur complementation. Hence, at every recursive level:

- The submatrices remain positive definite
- Their pivots remain non-zero

This guarantees safe recursion without pivoting.

Comparison with General Matrices

Matrix Type	**Pivoting Required**	**Reason**
General matrix	Yes	Zero or small pivots possible
Positive-definite matrix	No	Pivots always positive

Conclusion

Positive-definite matrices guarantee non-zero pivot elements,

hence LU decomposition can be performed safely without pivoting.

This property makes positive-definite matrices especially important in numerical algorithms, optimization problems, and scientific computing.

■

8 BFS vs DFS for Augmenting Paths

Key Observation

Augmenting paths should be short for faster convergence.

Why BFS is Better

- Finds shortest augmenting paths
- Minimizes number of augmentations

■

9 Why Dijkstra Fails with Negative Weights

Core Assumption

Once a node is finalized, its distance never changes.

Violation

Negative edges can later reduce distances, breaking correctness.

Conclusion

Dijkstra's algorithm cannot handle negative weights.

■

10 Structure of the Symmetric Difference of Two Matchings

Introduction

Matchings are a fundamental concept in graph theory and play a key role in problems related to network flow, bipartite graphs, and combinatorial optimization.

This question asks us to prove an important structural property:

> *Every connected component of the symmetric difference of two matchings in a graph is either a path or an even-length cycle.*

Basic Definitions

Let $G = (V, E)$ be an undirected graph.

- A **matching** is a set of edges such that no two edges share a common vertex.
- Let M_1 and M_2 be two matchings in graph G.
- The **symmetric difference** of M_1 and M_2 is defined as:

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

Intuitively, the symmetric difference contains edges that belong to exactly one of the two matchings.

Key Observation

Since M_1 and M_2 are matchings:

- Each vertex is incident to at most one edge from M_1
- Each vertex is incident to at most one edge from M_2

Therefore, in the graph formed by $M_1 \oplus M_2$:

$$\boxed{\text{Each vertex has degree at most 2}}$$

This observation is the core of the proof.

Degree-Based Structural Analysis

Let us analyze the possible degrees of vertices in $M_1 \oplus M_2$:

- Degree 0: Isolated vertex (not part of any component)
- Degree 1: Endpoint of a path
- Degree 2: Internal vertex of a path or a cycle

No vertex can have degree greater than 2.

Possible Connected Components

Given the degree constraint, each connected component of $M_1 \oplus M_2$ must be one of the following:

Case 1: Path

If a connected component contains vertices of degree 1, it must be a path. Such a path alternates between edges of M_1 and M_2.

Case 2: Cycle

If all vertices in the component have degree 2, the component forms a cycle. Since edges alternate between M_1 and M_2, the cycle length must be even.

Why Odd-Length Cycles Are Impossible

In the symmetric difference:

- Adjacent edges must belong to different matchings
- Alternation requires pairs of edges

An odd-length cycle would force two adjacent edges to belong to the same matching, which contradicts the definition of a matching.

Formal Proof

Let C be a connected component of $M_1 \oplus M_2$.

- Each vertex in C has degree ≤ 2
- Therefore, C is either a path or a cycle
- If C is a cycle, edges must alternate between M_1 and M_2
- Hence, the cycle length must be even

Conclusion

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

Why This Result Is Important

- Forms the theoretical basis of augmenting path algorithms
- Used in bipartite matching and network flow problems
- Essential for correctness of matching improvement techniques

■

11 Class Co-NP

Introduction

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

This question asks us to:

> *Define the class Co-NP and explain the type of problems that belong to this class.*

Background: NP

A decision problem belongs to the class **NP** if:

- The answer is **YES**
- There exists a certificate (witness)
- The certificate can be verified in polynomial time

Example:

- SAT: Given a satisfying assignment, we can verify it efficiently

Definition of Co-NP

A decision problem belongs to the class **Co-NP** if:

Its complement problem belongs to NP

Equivalently:

- The answer is **NO**
- There exists a certificate for the NO answer
- This certificate can be verified in polynomial time

Intuitive Understanding

- NP focuses on efficiently verifiable **YES** answers
- Co-NP focuses on efficiently verifiable **NO** answers

Thus, Co-NP captures problems where proving incorrectness is easy.

Examples of Co-NP Problems

- **UNSAT**: Given a Boolean formula, prove that it is unsatisfiable
- **Composite Number**: Prove that a number is not prime
- **TAUTOLOGY**: Determine whether a Boolean formula is always true

In each case, a NO certificate can be checked efficiently.

Relationship Between NP and Co-NP

- It is known that:
$$P \subseteq NP \cap Co\text{-}NP$$
- It is unknown whether:
$$NP = Co\text{-}NP$$

This is a major open problem in complexity theory.

Conclusion

Co-NP is the class of decision problems whose NO instances can be verified in polynomial time.

■

12 Verification of Boolean Circuit Output using DFS

Introduction

Boolean circuits are commonly used to represent computations in complexity theory. Verifying the correctness of a circuit's output is an important problem related to NP verification.

This question asks us to explain:

> *How the correctness of a Boolean circuit evaluating to TRUE can be verified in polynomial time using Depth First Search (DFS).*

Boolean Circuit as a Graph

A Boolean circuit can be represented as a **Directed Acyclic Graph (DAG)** where:

- Nodes represent logic gates (AND, OR, NOT)
- Edges represent signal flow
- Input nodes have fixed Boolean values
- The output node produces the final result

Key Idea of Verification

Given that the output evaluates to TRUE:

- We want to verify whether this result is correct
- We do not recompute blindly
- Instead, we verify gate consistency

Role of Depth First Search (DFS)

DFS is used to traverse the circuit from the output gate to the input gates:

- Start DFS from the output node
- Recursively visit all input gates
- Verify correctness bottom-up

Verification Process

For each gate encountered during DFS:

- AND gate: output is TRUE if all inputs are TRUE
- OR gate: output is TRUE if at least one input is TRUE
- NOT gate: output is inverse of input

If all gates satisfy their logical constraints, the output is correct.

Time Complexity Analysis

- DFS visits each gate once
- Each edge is explored once

Thus:

$$\boxed{\text{Time Complexity} = O(V + E)}$$

Since the circuit size is polynomial in the input size, verification is polynomial time.

Conclusion

$$\boxed{\text{Boolean circuit correctness can be verified efficiently using DFS.}}$$

This establishes why Boolean circuit evaluation belongs to NP.

■

13 NP-Hardness of the 3-SAT Problem

Introduction

The Boolean satisfiability problem (SAT) is the first problem proven to be NP-Complete. 3-SAT is a restricted version of SAT where each clause has exactly three literals.

This question asks:

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

Definition of 3-SAT

In 3-SAT:

- The Boolean formula is in Conjunctive Normal Form (CNF)

- Each clause contains exactly three literals

Example:

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4 \vee x_5)$$

Understanding NP-Hardness

A problem is NP-Hard if:

- Every problem in NP can be reduced to it
- The reduction must run in polynomial time

Reduction from SAT to 3-SAT

It has been proven that:

- Any SAT instance can be transformed into an equivalent 3-SAT instance
- This transformation increases size only polynomially

The transformed 3-SAT formula is satisfiable if and only if the original SAT formula is satisfiable.

Key Implication

Since:

- SAT is NP-Complete
- SAT $\leq_p$ 3-SAT

It follows that:

$$\boxed{\text{3-SAT is NP-Hard}}$$

Additional Observation

3-SAT also belongs to NP because:

- A satisfying assignment can be verified in polynomial time

Hence, 3-SAT is actually **NP-Complete**.

Conclusion

$$\boxed{\text{3-SAT is NP-Hard (and NP-Complete).}}$$

■