ADSA Assignment

Krushna Kalyan Mohanty
Student ID: A125008

Department of Computer Science and Engineering
IIIT Bhubaneswar

January 4, 2026

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

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

Explanation

The recursive Heapify operation is used to maintain the heap property by comparing a node with its children and moving it down the heap if necessary.

At each recursive call, Heapify proceeds to at most one of the subtrees. In the worst case, the element moves to the larger child subtree, whose size is at most $\frac{2n}{3}$.

Hence, the recurrence relation for the running time of Heapify is:

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

Solving the Recurrence

We expand the recurrence iteratively:

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

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

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

The recursion terminates when the problem size reduces to a constant:

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

Taking logarithm on both sides:

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

Rearranging:

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

Since $\log(3/2)$ is a positive constant, we get:

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

Final Result

Substituting $k = O(\log n)$ into the expanded recurrence:

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

Conclusion

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

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

2 Q: 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.

Explanation

Consider a binary heap stored in an array of size n, where indexing starts from 1.

In an array representation of a binary heap:

- The parent of a node at index i is located at index $\left\lfloor \frac{i}{2} \right\rfloor$
- The left child of a node at index i is located at index $2i$
- The right child of a node at index i is located at index $2i + 1$

A node is called a **leaf node** if it has no children.

Condition for Leaf Nodes

For a node at index i to have at least one child, the index of its left child must satisfy:

$$2i \leq n$$

If:

$$2i > n$$

then the node at index i has no children and is therefore a leaf node.

Dividing both sides by 2:

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

Index Range of Leaf Nodes

Thus, all nodes with indices satisfying:

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

are leaf nodes.

Since indices are integers, the leaf nodes are located at:

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

Conclusion

Hence, in an array of size n representing a binary heap, all leaf nodes are located at indices:

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

■

3

Question 3(a)

Show that in any heap containing n elements, the number of nodes at height h is at most $\frac{n}{2^{h+1}}$.

Proof

Consider a binary heap containing n elements. A binary heap is a complete binary tree, where all levels are completely filled except possibly the last level, which is filled from left to right.

Let the height of a node be defined as the number of edges on the longest downward path from that node to a leaf.

Observation

- The root of the heap has the maximum height.
- As we move down one level in the heap, the number of nodes doubles.
- Conversely, as height increases, the number of nodes decreases exponentially.

At height h, each node can have at most 2^h descendants below it. Therefore, the total number of nodes in the heap satisfies:

$$n \geq (\text{number of nodes at height } h) \times 2^{h+1}$$

Rearranging the inequality:

$$\text{Number of nodes at height } h \leq \frac{n}{2^{h+1}}$$

Conclusion

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

$$\boxed{\frac{n}{2^{h+1}}}$$

■

Question 3(b)

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

Proof

The Build-Heap algorithm constructs a heap by calling the `Heapify` procedure on all non-leaf nodes, starting from the lowest level up to the root.

Key Observations

- Leaf nodes do not require Heapify.
- A node at height h takes $O(h)$ time for Heapify.
- From part (a), the number of nodes at height h is at most $\frac{n}{2^{h+1}}$.

Total Time Complexity

The total time required by Build-Heap is the sum of the Heapify costs over all heights:

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

Factoring out n:

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

The series:

$$\sum_{h=0}^{\infty} \frac{h}{2^{h+1}}$$

is a convergent series and evaluates to a constant.

Therefore:

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

Conclusion

Hence, using the result from part (a), we conclude that the time complexity of the Build-Heap algorithm is:

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

■

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

Explanation

LU decomposition is a matrix factorization technique in which a given square matrix A is decomposed 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

LU decomposition is commonly used to solve systems of linear equations efficiently, especially when the same coefficient matrix is used with multiple right-hand sides.

Step-by-Step Procedure Using Gaussian Elimination

Consider a system of linear equations represented in matrix form:

$$AX = B$$

where A is an $n \times n$ matrix.

Step 1: Gaussian Elimination

Gaussian Elimination is applied to matrix A to eliminate the elements below the main diagonal. This process transforms A into an upper triangular matrix U.

At each elimination step:

- A pivot element is chosen on the diagonal
- Multipliers are used to eliminate entries below the pivot

Step 2: Formation of the Lower Triangular Matrix L

The multipliers used during the elimination process are stored in matrix L.

Properties of matrix L:

- All diagonal elements of L are equal to 1
- Elements below the diagonal contain the elimination multipliers
- Elements above the diagonal are zero

Step 3: Formation of the Upper Triangular Matrix U

After elimination:

- The resulting matrix becomes the upper triangular matrix U
- All elements below the diagonal are zero

Thus, the original matrix can be written as:

$$A = LU$$

Step 4: Solving the System Using LU Decomposition

Once A is decomposed:

$$AX = B \Rightarrow LUX = B$$

The solution is obtained in two stages:

1. Solve $LY = B$ using forward substitution
2. Solve $UX = Y$ using backward substitution

Advantages of LU Decomposition

- Reduces computational cost for solving multiple systems
- More efficient than repeated Gaussian elimination
- Widely used in numerical linear algebra

Conclusion

LU decomposition using Gaussian Elimination factorizes a matrix into lower and upper triangular matrices. This decomposition simplifies the solution of linear systems and improves computational efficiency.

$$\boxed{A = LU}$$

■

5 Q: 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 main summations. We evaluate each part step by step.

First Summation

Consider:

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

The inner summation:

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

Thus, the first summation becomes:

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

Second Summation

Now consider:

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

The inner summation:

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

Hence, the second summation becomes:

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

Combining Both Parts

Combining the two simplified summations:

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

Both summations evaluate to:

$$O\left(\sum_{i=1}^{n} i \right) = O(n^2)$$

Final Result

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

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

■

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

Proof

Let A be a square matrix partitioned as:

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

where B is a square and invertible submatrix.

The **Schur complement** of B in A is defined as:

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

Key Result

For a block matrix A with invertible block B, the determinant of A satisfies:

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

Non-Singularity Argument

Since A is given to be non-singular:

$$\det(A) \neq 0$$

Also, since B is invertible:

$$\det(B) \neq 0$$

From the determinant relation:

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

Substituting the above results:

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

This implies:

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

Hence, the Schur complement S is non-singular.

Conclusion

Therefore, if the matrix A is non-singular and the block B is invertible, then its Schur complement is also non-singular.

$$\boxed{\text{Schur complement of a non-singular matrix is non-singular}}$$

■

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

Proof

Let A be a real symmetric positive-definite matrix of order n. By definition, a matrix A is **positive-definite** if:

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

Properties of Positive-Definite Matrices

Positive-definite matrices satisfy the following important properties:

- All leading principal minors of A are positive.
- All diagonal elements of A are strictly positive.
- The matrix A is non-singular.

LU Decomposition without Pivoting

In LU decomposition using Gaussian elimination, division by zero can occur if a pivot element becomes zero. To avoid this, pivoting is generally required.

However, for a positive-definite matrix:

- The first pivot element $a_{11} > 0$.
- After each elimination step, the remaining submatrix (Schur complement) is also positive-definite.
- Therefore, all subsequent pivot elements are strictly positive.

Since no pivot element becomes zero, division by zero never occurs during the elimination process.

Recursive Strategy Argument

At each recursive step of LU decomposition:

- The pivot corresponds to a leading principal minor.
- For positive-definite matrices, all leading principal minors are non-zero.

Hence, the recursive LU decomposition proceeds safely without requiring row exchanges.

Conclusion

Since positive-definite matrices have strictly positive pivots at every stage of Gaussian elimination, LU decomposition can be performed without pivoting and without the risk of division by zero.

$$\boxed{\text{Positive-definite matrices admit LU decomposition without pivoting}}$$

■

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

Answer and Justification

An **augmenting path** is a path that starts and ends at free vertices and alternates between unmatched and matched edges. Augmenting paths are used in matching algorithms to increase the size of a matching.

Choice of Search Strategy

Breadth First Search (BFS) is preferred over Depth First Search (DFS) for finding augmenting paths.

Justification

- BFS explores the graph level by level and always finds the **shortest augmenting path**.
- Shorter augmenting paths reduce the number of augmentations required to reach a maximum matching.
- BFS improves efficiency and guarantees better overall time complexity.
- The **Hopcroft–Karp algorithm** for bipartite matching explicitly uses BFS to find multiple shortest augmenting paths simultaneously.

Why DFS Is Not Preferred

- DFS may find longer augmenting paths unnecessarily.
- Longer paths can lead to inefficient performance.
- DFS does not guarantee minimal path length.

Conclusion

Therefore, Breadth First Search (BFS) is the preferred method for finding augmenting paths in a graph due to its ability to find the shortest augmenting paths efficiently.

$$\boxed{\text{BFS is preferred over DFS for finding augmenting paths}}$$

■

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

Explanation

Dijkstra's algorithm is a greedy algorithm used to find the shortest path from a source vertex to all other vertices in a weighted graph.

The fundamental assumption of Dijkstra's algorithm is that **all edge weights are non-negative**.

Key Assumption of Dijkstra's Algorithm

At each step, Dijkstra's algorithm:

- Selects the vertex with the minimum tentative distance.
- Permanently marks this distance as the shortest possible distance.

This assumption is valid only when all edge weights are non-negative.

Effect of Negative Edge Weights

If the graph contains a negative edge weight:

- A vertex that has already been marked as having the shortest distance may later receive a shorter path through a negative edge.
- This violates the greedy choice property of Dijkstra's algorithm.

Illustrative Reasoning

Suppose a vertex u is selected with the minimum tentative distance. If there exists a negative edge from another vertex v to u, then a shorter path to u could be discovered later, contradicting the assumption that the distance to u was final.

Thus, Dijkstra's algorithm may produce incorrect shortest path results when negative edges are present.

Conclusion

Since Dijkstra's algorithm relies on the greedy assumption that once a vertex is selected its shortest distance is finalized, the presence of negative edge weights breaks this assumption.

Therefore, Dijkstra's algorithm cannot be applied to graphs with negative edge weights.

$$\boxed{\text{Dijkstra's algorithm fails for graphs with negative edge weights}}$$

■

10 Q: 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.

Proof

Let $G = (V, E)$ be a 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 all edges that belong to exactly one of the two matchings.

Degree of Vertices in $M_1 \oplus M_2$

Since M_1 and M_2 are matchings:

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

Therefore, in the symmetric difference $M_1 \oplus M_2$, a vertex can be incident to:

- At most one edge from M_1
- At most one edge from M_2

Hence, the degree of any vertex in $M_1 \oplus M_2$ is at most 2.

Structure of Connected Components

A graph in which every vertex has degree at most 2 can only have connected components that are:

- Simple paths (vertices of degree 1 at the ends)
- Cycles (all vertices of degree 2)

Thus, every connected component of $M_1 \oplus M_2$ is either a path or a cycle.

Parity of Cycles

Along any connected component, the edges alternate between M_1 and M_2.

In a cycle:

- The alternation must return to the starting matching
- This is possible only if the cycle contains an even number of edges

Therefore, all cycles in $M_1 \oplus M_2$ are of even length.

Conclusion

Hence, every connected component of the symmetric difference of two matchings in a graph G is either:

- A path, or
- An even-length cycle

$$\boxed{\text{Each connected component is a path or an even-length cycle}}$$

■

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

Definition of Co-NP

The complexity class **Co-NP** consists of all decision problems whose **complements** belong to the class NP.

Formally,

$$\text{Co-NP} = \{L \mid \overline{L} \in \text{NP}\}$$

where $\overline{L}$ denotes the complement of language L.

Explanation

A problem belongs to Co-NP if, whenever the answer is **NO**, there exists a certificate that can be verified in polynomial time.

This is in contrast to NP, where a **YES** answer can be verified in polynomial time.

Type of Problems in Co-NP

Problems in Co-NP are typically:

- Problems where proving **non-existence** is easier than proving existence.
- Problems whose incorrect solutions can be efficiently verified.

Examples of Co-NP Problems

- **TAUT**: Given a Boolean formula, determine whether it is a tautology.
- **UNSAT**: Determine whether a Boolean formula is unsatisfiable.

For example, a Boolean formula is unsatisfiable if there exists a proof showing that no assignment satisfies it, which can be verified in polynomial time.

Relationship with NP

It is known that:

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

However, it is an open problem in computer science whether:

$$NP = \text{Co-NP}$$

Conclusion

Thus, Co-NP contains decision problems for which a **NO** answer can be verified efficiently. These problems are complements of NP problems and play an important role in complexity theory.

$$\boxed{\text{Co-NP consists of problems whose NO instances are verifiable in polynomial time}}$$

■

12 Q: 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).

Explanation

A Boolean circuit consists of:

- Input nodes (Boolean variables)
- Logic gates such as AND, OR, and NOT
- A single output node

The circuit can be represented as a **directed acyclic graph (DAG)**, where edges represent the flow of computation from inputs to the output.

Verification Strategy

Given that the output of the circuit evaluates to **true**, the correctness of this result can be verified using a Depth First Search (DFS) traversal of the circuit graph.

Steps Involved in Verification

1. Start DFS from the output node of the circuit.
2. Recursively traverse all input edges of the current gate.
3. At each gate, verify whether its output value is consistent with:
    - The type of the gate (AND, OR, NOT)
    - The values of its input gates
4. Continue this process until all input variables are reached.

Correctness Checking

During DFS traversal:

- Each gate is visited exactly once.
- The output value of each gate is recomputed from its inputs.
- If all recomputed values match the given values, the circuit evaluation is correct.

Time Complexity Analysis

Let:

- V be the number of gates (vertices)
- E be the number of connections (edges)

DFS runs in:

$$O(V + E)$$

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

Conclusion

Thus, using Depth First Search, the correctness of a Boolean circuit whose output evaluates to true can be verified efficiently in polynomial time.

$$\boxed{\text{Boolean circuit verification can be done in polynomial time using DFS}}$$

■

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

Explanation

The **3-SAT problem** is a decision problem in which a Boolean formula is given in **conjunctive normal form (CNF)**, where each clause contains exactly three literals. The objective is to determine whether there exists a truth assignment that satisfies the formula.

3-SAT Belongs to NP

Given a truth assignment to the variables:

- Each clause can be checked in constant time.
- All clauses can be verified in polynomial time.

Hence, 3-SAT belongs to the class NP.

NP-Hardness of 3-SAT

To prove that 3-SAT is NP-Hard, we show that a known NP-Complete problem can be reduced to 3-SAT in polynomial time.

The **SAT problem** (Boolean satisfiability) is known to be NP-Complete. Any Boolean formula in CNF can be transformed into an equivalent 3-CNF formula by:

- Breaking long clauses into clauses of size three using additional variables.
- Preserving satisfiability during the transformation.

This transformation can be performed in polynomial time.

Conclusion of Reduction

Since:

- SAT reduces to 3-SAT in polynomial time, and
- SAT is NP-Complete,

it follows that 3-SAT is NP-Hard.

Final Conclusion

Thus, 3-SAT is both:

- In NP, and
- NP-Hard

Therefore, the 3-SAT problem is NP-Complete.

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

■

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

Explanation

The **2-SAT problem** is a special case of the Boolean satisfiability problem in which each clause of the Boolean formula contains at most two literals. The objective is to determine whether there exists a truth assignment that satisfies the formula.

Is 2-SAT NP-Hard?

The 2-SAT problem is **not NP-Hard**.

Unlike 3-SAT, the restriction to at most two literals per clause significantly reduces the complexity of the problem. There is no known polynomial-time reduction from an NP-Complete problem to 2-SAT unless P = NP.

Therefore, 2-SAT does not belong to the class of NP-Hard problems.

Polynomial-Time Solvability of 2-SAT

The 2-SAT problem can be solved in polynomial time using the **implication graph** method.

Implication Graph Approach

Each clause of the form:

$$(x \lor y)$$

is converted into two implications:

$$(\neg x \Rightarrow y) \quad \text{and} \quad (\neg y \Rightarrow x)$$

The resulting implication graph is a directed graph whose vertices represent literals.

Strongly Connected Components

The formula is satisfiable if and only if:

- No variable and its negation belong to the same strongly connected component (SCC).

Strongly connected components can be computed using:

- Kosaraju's algorithm, or
- Tarjan's algorithm

both of which run in linear time.

Time Complexity

Let:

- V be the number of literals
- E be the number of implications

The total running time is:

$$O(V + E)$$

which is polynomial in the size of the input.

Conclusion

Thus:

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

$$\boxed{\text{2-SAT} \in \text{P}}$$

$\blacksquare$