ADSA Assignment

MTech (CS), 1st Sem, 2025

IIIT Bhubaneswar

Student ID: A125019
Student Name: Shaikh Jamil Ahemad

Solutions of ADSA Assignment

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

Solution (Derivation):

Given the recurrence relation:

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

Using the substitution method, we expand the recurrence as follows:

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

The recursion terminates when:

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

Taking logarithms on both sides:

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

Solution (Conclusion):

Substituting this value of k:

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

Since $T(1) = O(1)$, we conclude:

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

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

Solution:

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

For any node at index i, the indices of its children are:

$$\text{Left child} = 2i, \quad \text{Right child} = 2i + 1$$

A node is a leaf if it has no children, i.e.,

$$2i > n$$

This implies:

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

Hence, all indices satisfying:

$$i = \left\lfloor \frac{n}{2} \right\rfloor + 1, \dots, n$$

correspond to leaf nodes.

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

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

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

Solution:

In a binary heap, the height of a node is defined as the number of edges on the longest downward path from that node to a leaf.

A node at height h has at least 2^h descendants (including itself) in the heap.

Since the total number of nodes in the heap is n, the maximum possible number of nodes at height h is bounded by:

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

Because only internal nodes can have height greater than or equal to 1, the number of nodes at height h is at most:

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

Therefore, in any heap of size n, 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:

The `Build-Heap` algorithm calls `Heapify` on all non-leaf nodes of the heap, starting from the lowest level up to the root.

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

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

The time required to `Heapify` a node at height h is $O(h)$.

Hence, the total time complexity of `Build-Heap` is:

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

Ignoring constant factors, we get:

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

Since the series $\sum_{h=0}^{\infty} \frac{h}{2^h}$ converges to a constant, we obtain:

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

Therefore, the time complexity of the `Build-Heap` algorithm is $O(n)$.

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

Solution:

LU decomposition factors a square matrix A into the product:

$$A = LU$$

where L is a lower triangular matrix with unit diagonal entries and U is an upper triangular matrix.

The decomposition is obtained using Gaussian Elimination as follows:

(a) Apply Gaussian Elimination to eliminate elements below the main diagonal.

(b) Store the elimination multipliers in the corresponding positions of matrix L.

(c) The resulting upper triangular matrix forms matrix U.

(d) Set all diagonal entries of L to 1.

Algorithm (LU Decomposition using Gaussian Elimination):

```
1: Input: Square matrix $A \in \mathbb{R}^{n\times n}$
2: Output: Lower triangular matrix $L$ and upper triangular matrix $U$
3: Initialize $L = I_n$, $U = A$
4: for $k = 1$ to $n - 1$ do
5:     for $i = k + 1$ to $n$ do
6:         $L_{ik} \leftarrow \frac{U_{ik}}{U_{kk}}$
7:         for $j = k$ to $n$ do
8:             $U_{ij} \leftarrow U_{ij} - L_{ik} \cdot U_{kj}$
9:         end for
10:    end for
11: end for
12: Return $L, U$
```

Thus, Gaussian Elimination systematically produces the LU decomposition of matrix A.

5. 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 is:

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

Consider the first summation. For a fixed i, the inner sum $\sum_{j=1}^{i-1} O(1)$ performs $(i-1)$ constant-time operations, hence:

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

Therefore:

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

Similarly, in the second summation, the inner loop executes $(n-i)$ constant-time operations for each i, giving:

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

Hence:

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

Combining both results:

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

Both summations are arithmetic series and evaluate to $O(n^2)$. Therefore:

$$\therefore \quad T(n) = O(n^2)$$

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

Solution:

Let the non-singular matrix A be partitioned as:

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

where B is a non-singular square matrix.

The Schur complement of B in A is defined as:

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

Assume, for the sake of contradiction, that the Schur complement S is singular. Then there exists a non-zero vector y such that:

$$Sy = 0$$

Define a vector x as:

$$x = \begin{bmatrix} -B^{-1}Cy \\ y \end{bmatrix}$$

Since $y \neq 0$, we have $x \neq 0$.

Now consider:

$$Ax = \begin{bmatrix} B & C \\ D & E \end{bmatrix} \begin{bmatrix} -B^{-1}Cy \\ y \end{bmatrix} = \begin{bmatrix} -Cy + Cy \\ -DB^{-1}Cy + Ey \end{bmatrix} = \begin{bmatrix} 0 \\ Sy \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Thus, $Ax = 0$ for some non-zero vector x, which contradicts the assumption that A is non-singular.

Therefore, the Schur complement S must be non-singular.

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

7. 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:

Let A be a symmetric positive-definite matrix. Then for any non-zero vector x,

$$x^T Ax > 0$$

In the recursive LU decomposition strategy, at each step we partition A as:

$$A = \begin{bmatrix} \alpha & v^T \\ v & B \end{bmatrix}$$

where α is the leading principal element.

Since A is positive-definite, all its leading principal minors are positive. In particular,

$$\alpha > 0$$

Therefore, division by α during Gaussian Elimination is always valid, and no pivoting is required.

The Schur complement of α in A is:

$$S = B - \frac{1}{\alpha} v v^T$$

From properties of positive-definite matrices, the Schur complement S is also positive-definite.

Thus, at each recursive step:

- the pivot element is non-zero and positive,
- the remaining subproblem remains positive-definite.

Hence, LU decomposition can proceed without pivoting and without encountering division by zero.

Therefore, positive-definite matrices are suitable for LU decomposition and do not require pivoting in the recursive strategy.

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

Solution:

For finding an augmenting path, Breadth First Search (BFS) should be applied.

BFS finds the shortest augmenting path in terms of number of edges. Using shortest augmenting paths ensures that the matching size increases monotonically and leads to a polynomial-time algorithm.

In particular, the Hopcroft–Karp algorithm for bipartite matching uses BFS to find layers of shortest augmenting paths, which significantly improves the time complexity.

Depth First Search (DFS) may find arbitrarily long augmenting paths and can lead to inefficient performance.

Therefore, BFS is preferred over DFS for finding augmenting paths in a graph.

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

Solution:

Dijkstra's algorithm assumes that once a vertex is extracted from the priority queue with the minimum tentative distance, its shortest-path distance is final.

This assumption holds only when all edge weights are non-negative. With non-negative weights, any alternative path to an already processed vertex must be longer.

If negative edge weights are present, a shorter path to an already processed vertex may be found later through a negative-weight edge.

Hence, the greedy choice made by Dijkstra's algorithm can be invalidated, leading to incorrect shortest-path results.

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

10. 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:

Let M_1 and M_2 be two matchings in graph G. Consider their symmetric difference:

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

In the subgraph induced by $M_1 \oplus M_2$, each vertex has degree at most 2, since a vertex can be incident to at most one edge from each matching.

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

Moreover, edges in each component alternate between M_1 and M_2. Hence, any cycle must contain an equal number of edges from M_1 and M_2, implying that the cycle has even length.

Thus, every connected component of the symmetric difference of two matchings is either a path or an even-length cycle.

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

Solution:

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

Formally, a language L is in **Co-NP** if and only if its complement $\overline{L}$ is in **NP**.

Equivalently, a problem is in **Co-NP** if for every input instance for which the answer is `NO`, there exists a certificate that can be verified in polynomial time.

Problems in **Co-NP** typically involve verifying that no solution exists. A classic example is the `TAUT` problem, which asks whether a Boolean formula is true for all possible assignments.

Thus, **Co-NP** contains problems whose negative instances admit efficiently verifiable proofs.

12. 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:

A Boolean circuit can be represented as a directed acyclic graph (DAG), where each node corresponds to a logical gate and edges represent signal flow.

Given a certificate consisting of the input values, the correctness of the output can be verified by evaluating the circuit using Depth First Search (DFS).

DFS is initiated from the output gate and recursively evaluates all input gates before computing the value of the current gate.

Algorithm (DFS-Based Circuit Verification):

```
1: Input: Boolean circuit $C$, input assignment $x$
2: Output: true if circuit evaluates correctly, else false
3: Perform DFS starting from the output gate
4: for each gate $g$ visited during DFS do
5:     if $g$ is an input gate then
6:         Assign its value from $x$
7:     else
8:         Compute the output of $g$ based on its gate type
9:     end if
10: end for
11: Return the value computed at the output gate
```

Since the circuit is acyclic, each gate is visited at most once. Therefore, the verification runs in time linear in the size of the circuit, which is polynomial in the input size.

Hence, the correctness of the Boolean circuit output can be verified in polynomial time using DFS.

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

Solution:

The **3-SAT** problem consists of determining the satisfiability of a Boolean formula in conjunctive normal form where each clause has exactly three literals.

Clearly, **3-SAT** is in **NP**, since given a truth assignment to the variables, the formula can be evaluated in polynomial time.

To prove that **3-SAT** is NP-Hard, we show a polynomial-time reduction from **SAT** to **3-SAT**.

Let ϕ be an arbitrary Boolean formula in CNF. Each clause of ϕ can be transformed into an equivalent set of clauses, each containing exactly three literals, by introducing a linear number of new variables. This transformation preserves satisfiability and can be performed in polynomial time.

Hence,

$$\textbf{SAT} \leq_p \textbf{3-SAT}.$$

Since **SAT** is NP-Complete, it follows that **3-SAT** is NP-Hard.

Therefore, the **3-SAT** problem is NP-Hard (and in fact NP-Complete).

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

Solution:

The **2-SAT** problem consists of determining the satisfiability of a Boolean formula in conjunctive normal form where each clause has at most two literals.

Unlike 3-SAT, the **2-SAT** problem is not NP-Hard and can be solved in polynomial time.

Each clause of the form $(a \lor b)$ can be rewritten as the implications:

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

Using this transformation, a 2-SAT instance can be represented as an *implication graph*, where each literal is a vertex and each implication is a directed edge.

The formula is satisfiable if and only if, for every variable x, the literals x and $\neg x$ do not belong to the same strongly connected component (SCC) of the implication graph.

Strongly connected components can be computed using Depth First Search (DFS) in linear time.

Since the implication graph has $O(n)$ vertices and edges, the overall algorithm runs in polynomial time.

Therefore, **2-SAT** is solvable in polynomial time and is not NP-Hard (unless $\mathbf{P} = \mathbf{NP}$).