ADSA Assignment

Student ID: A125006
Name: Hari Shankar Gharai

MTech (CS), 1st Sem, 2025
IIIT Bhubaneswar

Problem 1

Prove that the time complexity of the recursive Heapify operation is $O(\log n)$ using the recurrence relation $T(n) = T(2n/3) + O(1)$.

Solution:(Master Theorem)

Instead of the substitution method, we can apply the **Master Theorem** for recurrences of the form $T(n) = aT(n/b) + f(n)$.

Given the recurrence:

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

We identify the parameters:

- $a = 1$ (since there is 1 recursive call)
- $b = 3/2$ (since the problem size reduces to $2n/3$, which is $n/(3/2)$)
- $f(n) = \Theta(1)$ (constant work outside recursion)

We calculate the critical exponent:

$$\log_b a = \log_{3/2} 1 = 0$$

Comparing $f(n)$ with $n^{\log_b a}$:

$$f(n) = \Theta(1) = \Theta(n^0) = \Theta(n^{\log_b a})$$

This matches **Case 2** of the Master Theorem. Therefore:

$$T(n) = \Theta(n^{\log_b a} \log n) = \Theta(1 \cdot \log n) = O(\log n)$$

Problem 2

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

Solution:(Counting Internal Nodes)

Let the heap be stored in an array $A[1 \dots n]$. A node at index i is an *internal node* if and only if it has at least one child. In a binary heap, the left child of a node i is located at index $2i$.

For a node i to be an internal node, its left child must exist within the array bounds:

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

Thus, the indices $1, 2, \dots, \lfloor n/2 \rfloor$ represent all the internal nodes.

Since every node in the heap is either an internal node or a leaf, the remaining nodes must be leaves. The set of remaining indices is:

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

Therefore, all leaves are located in this range.

Problem 3

(a) Show that the number of nodes at height h is at most $\lceil n/2^{h+1} \rceil$. (b) Prove Build-Heap is $O(n)$.

Solution

(a) Structural Property: Instead of counting descendants, observe that in a binary heap, approximately half the nodes are leaves (height 0), a quarter are height 1, etc. Specifically, if we remove all leaves, the nodes at height h in the original tree become nodes at height $h-1$ in the new tree. Let N_h be the number of nodes at height h. We can observe that $N_h \approx N_{h-1}/2$. Rigourously, the number of nodes of height h is bounded by $\lceil n/2^{h+1} \rceil$ due to the binary tree structure halving the population at each level up.

(b) Convergence of Series: The cost of `Build-Heap` is the sum of costs of `Heapify` at each level.

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

To evaluate the sum $S = \sum_{h=0}^{\infty} \frac{h}{2^h}$, consider the geometric series $f(x) = \sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$ for $|x| < 1$. Differentiating with respect to x:

$$f'(x) = \sum_{k=1}^{\infty} k x^{k-1} = \frac{1}{(1-x)^2}$$

Multiplying by x:

$$x f'(x) = \sum_{k=1}^{\infty} k x^k = \frac{x}{(1-x)^2}$$

Substitute $x = 1/2$:

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = \frac{1/2}{1/4} = 2$$

Thus, the summation converges to a constant.

$$T(n) = O(n \cdot 2) = O(n)$$

Problem 4

Explain LU decomposition using Gaussian Elimination.

Solution:(Elementary Matrices)

Gaussian elimination can be viewed as a sequence of matrix multiplications. To eliminate the entries below the diagonal in the k-th column of matrix A, we multiply A on the left by an *elementary lower triangular matrix* M_k.

$$M_{n-1} \dots M_2 M_1 A = U$$

where U is the resulting upper triangular matrix (row echelon form).

Each M_k is lower triangular with unit diagonals. The inverse of such a matrix, $L_k = M_k^{-1}$, is also lower triangular and is obtained simply by flipping the signs of the off-diagonal multipliers. We can rewrite the equation as:

$$A = (M_1^{-1} M_2^{-1} \dots M_{n-1}^{-1})U$$

$$A = (L_1 L_2 \dots L_{n-1})U$$

Let $L = L_1 L_2 \dots L_{n-1}$. The product of lower triangular matrices with unit diagonals is itself a lower triangular matrix with unit diagonals. Thus, we arrive at the decomposition $A = LU$. The entries of L below the diagonal are exactly the multipliers used during Gaussian elimination.

Problem 5

Solve the recurrence arising from LUP decomposition: $T(n) \approx \sum_{i=1}^{n}[i + (n - i)]$**.**

Solution:(Geometric Interpretation)

The recurrence describes the work done in a nested loop structure.

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

Let's simplify the inner logic. The term $\sum_{j=1}^{i-1} O(1)$ represents the work done on the "left" part of the row, which scales with i. The term $\sum_{j=i+1}^{n} O(1)$ represents work on the "right" part, scaling with $n - i$.

The total work is simply the sum of work over all rows $i = 1$ to n. For each row i, the algorithm processes the entire row of length n (parts $1 \dots i-1$ and $i+1 \dots n$). Thus, for every iteration i, the work is $O(n)$. Summing this over n iterations:

$$T(n) = \sum_{i=1}^{n} O(n) = n \cdot O(n) = O(n^2)$$

Problem 6

Prove that if A is non-singular, its Schur complement is non-singular.

Solution:(Determinant Identity)

Let A be partitioned as $A = \begin{pmatrix} B & C \\ D & E \end{pmatrix}$. The Schur complement of B in A is $S = E - DB^{-1}C$.

There exists a known block-decomposition of the determinant for partitioned matrices:

$$\det(A) = \det(B) \cdot \det(E - DB^{-1}C)$$

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

We are given that A is non-singular, which implies $\det(A) \neq 0$. We assume the submatrix B is non-singular (required to form S), so $\det(B) \neq 0$. Since the product $\det(B) \cdot \det(S)$ is non-zero, it must be that $\det(S) \neq 0$. A non-zero determinant implies that the matrix S is non-singular.

Problem 7

Prove positive-definite matrices are suitable for LU decomposition without pivoting.

Solution:(Energy Argument)

A matrix A is positive definite if $x^T Ax > 0$ for all non-zero vectors x. In the first step of Gaussian elimination, the pivot element is a_{11}. Choose the unit vector $e_1 = [1, 0, \ldots, 0]^T$. Then $e_1^T A e_1 = a_{11}$. Since A is positive definite, $a_{11} > 0$. Thus, the first pivot is non-zero, and we can proceed without swapping rows.

After one step of elimination, we obtain a Schur complement S for the remaining $(n-1) \times (n-1)$ submatrix. A key property of positive definite matrices is that their Schur complements are also positive definite. By induction, every subsequent pivot element (which is the leading entry of the current Schur complement) will be positive. Since all pivots are strictly positive, they are never zero, guaranteeing that the algorithm never encounters a division by zero.

Problem 8

Augmenting Path: BFS or DFS?

Solution:(Complexity Perspective)

Breadth First Search (BFS) is strictly preferred over Depth First Search (DFS) for finding augmenting paths in Maximum Bipartite Matching (e.g., Edmonds-Karp or Hopcroft-Karp).

If we use DFS, we might select a very long path that snakes through the graph unnecessarily. Augmenting along such a path might only increase the matching size by 1 while modifying many edges, potentially "blocking" shorter, more effective paths. Using DFS, the algorithm is equivalent to the Ford-Fulkerson method with arbitrary path selection, which can have poor pseudo-polynomial time complexity in general flow networks.

In contrast, BFS guarantees finding the **shortest** augmenting path (fewest edges). It has been proven (Edmonds-Karp analysis) that if one always augments along the shortest path, the total number of augmentations is bounded polynomially ($O(VE)$), leading to a much more efficient algorithm ($O(VE^2)$ for general max flow, or $O(E\sqrt{V})$ for Hopcroft-Karp).

Problem 9

Why Dijkstra's algorithm fails with negative edge weights.

Solution:(Counter-Example)

Dijkstra's algorithm is a greedy approach that assumes "optimal substructure" grows monotonically: extending a path cannot make it shorter. Negative edges violate this.

Consider the graph:

- $A \to B$ (weight 5)
- $A \to C$ (weight 2)
- $C \to B$ (weight -10)

1. Start at A. Distance to $A = 0$.
2. Relax neighbors: $D(B) = 5, D(C) = 2$.
3. Dijkstra picks the node with the smallest distance: C (dist 2). C is marked "visited" and finalized.
4. Relax neighbors of C: $C \to B$ updates B to $2 + (-10) = -8$.
5. Now Dijkstra picks B.

If the graph were slightly more complex, say B connects to a destination D, Dijkstra might have already visited and finalized B before discovering the path via C if the edge $A \to B$ was very short and C was far, but a massive negative edge connected them. The core issue: Dijkstra finalized node B assuming 5 was optimal, but a negative edge later revealed a path of cost -8. Dijkstra does not revisit finalized nodes, leading to an incorrect result.

Problem 10

Connected components of Symmetric Difference of Matchings.

Solution:(Graph Traversal Argument)

Let $G' = (V, M_1 \oplus M_2)$. The edges in G' belong to either M_1 or M_2, but not both. Since M_1 and M_2 are matchings, no vertex in G' can be incident to more than one edge from M_1 and more than one edge from M_2. Thus, the maximum degree of any vertex in G' is 2.

Consider walking starting from an arbitrary vertex v in a component. If $\deg(v) = 1$, we must follow the single edge. At the next vertex, if the degree is 2, we must exit via the edge belonging to the matching we didn't just arrive on (alternating). We continue this path. Since the graph is finite, the path must either:

1. Terminate at a vertex with degree 1 (forming a simple Path).
2. Loop back to the start vertex (forming a Cycle).

If it forms a cycle, the edges must alternate $M_1, M_2, M_1, \ldots$. For the cycle to close back to the start node consistent with matching constraints, it must enter the start node via a different matching type than it left. This implies the cycle must have an even number of edges.

Problem 11

Define Co-NP and explain.

Solution:(Verification of Counter-Examples)

Definition: Co-NP is the class of decision problems where the "No" instances can be efficiently verified. Formally, $L \in$ Co-NP if there exists a polynomial-time verifier $V(x, c)$ such that:

$$x \notin L \iff \exists c \text{ (certificate) such that } V(x, c) = \text{accept}$$

Explanation: While NP asks "Does there exist a solution?", Co-NP asks "Are all cases valid?" (which is equivalent to asking "Does there NOT exist a counter-example?"). For example, the problem PRIME (is N prime?) was long considered in Co-NP before being found to be in P. The "No" instance (i.e., N is composite) is easily verified by providing a factor as a certificate. Another example: GRAPH NON-ISOMORPHISM. To prove two graphs are *not* isomorphic, one might provide a specific invariant (like node degrees) that differs, acting as a certificate for the "No" instance of Isomorphism.

Problem 12

Verify Boolean Circuit (True output) with DFS.

Solution:(Topological Evaluation)

A Boolean circuit is a Directed Acyclic Graph (DAG). To verify an output, we simply need to simulate the circuit. The most natural way to evaluate a DAG is via a **Topological Sort** or a Post-Order Traversal.

1. Perform a topological sort of the gates (or ensure we process inputs before outputs).
2. Iterate through the sorted gates. For every gate g_i, the values of its inputs have already been computed.
3. Compute the value of g_i in $O(1)$ time and store it.
4. The final result is the value at the output gate.

DFS naturally performs a post-order traversal. When DFS returns from children (inputs to the gate), their values are ready. The current node computes its logic (AND/OR/NOT) and returns the result up the recursion stack. Since the graph size is N, and we visit each edge once, the complexity is $O(V + E)$, which is polynomial.

Problem 13

Is 3-SAT NP-Hard? Justify.

Solution:(Reduction Mechanics)

Yes, 3-SAT is NP-Hard. We justify this by outlining the reduction $SAT \leq_p 3\text{-}SAT$. The core difficulty is transforming a clause with $k > 3$ literals into clauses of size 3 without changing satisfiability. Given a clause $C = (l_1 \vee l_2 \vee \cdots \vee l_k)$, we introduce new auxiliary variables $z_1, \ldots, z_{k-3}$. We replace C with the chain of clauses:

$$(l_1 \vee l_2 \vee z_1) \wedge (\neg z_1 \vee l_3 \vee z_2) \wedge (\neg z_2 \vee l_4 \vee z_3) \wedge \cdots \wedge (\neg z_{k-3} \vee l_{k-1} \vee l_k)$$

This transformation acts like a "bucket brigade". If the original clause is true, at least one literal l_i is true. We can set the z variables to propagate the "True" value through the chain to satisfy all constructed clauses. This reduction runs in polynomial time. Since SAT is NP-Hard (Cook-Levin), and we can reduce it to 3-SAT, 3-SAT is also NP-Hard.

Problem 14

Is 2-SAT NP-Hard?

Solution:(Implication Graph Logic)

2-SAT is **not** NP-Hard; it is in P. We can model the problem using an **Implication Graph**. A clause $(A \vee B)$ is equivalent to $(\neg A \implies B)$ and $(\neg B \implies A)$. We build a graph where nodes are literals. Edges represent these implications. **Theorem:** The formula is unsatisfiable if and only if there exists a variable x such that there is a path from x to $\neg x$ AND a path from $\neg x$ to x. This condition implies x and $\neg x$ belong to the same **Strongly Connected Component (SCC)**. We can find SCCs using Tarjan's algorithm or Kosaraju's algorithm in $O(V + E)$ time. If no variable shares an SCC with its negation, a satisfying assignment can be constructed by topologically sorting the component graph. Thus, 2-SAT is linear time solvable.