ADSA Assignment Solutions

Priyanshu Kumar
Registration No: A125015

January 4, 2026

1 Time Complexity of Recursive Heapify

Problem Statement: Prove that the time complexity of the recursive *Heapify* operation is $O(\log n)$, given the recurrence:

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

Explanation: In a binary heap, the Heapify operation compares a node with its children and swaps it with the largest (or smallest) child if the heap property is violated. After the swap, Heapify is recursively called on only one subtree. At each recursive call:

- The problem size reduces from n to at most $\frac{2n}{3}$.
- The work done at each step is constant, i.e., $O(1)$.

Solving the Recurrence:

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

The recursion terminates when:

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

Taking logarithms:

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

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

Final Result:

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

Thus, the recursive Heapify operation runs in logarithmic time.

2 Location of Leaf Nodes in a Binary Heap

Claim 1. In an array of size n representing a binary heap, all leaf nodes are located at indices:

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

Explanation: In a binary heap stored as an array (1-indexed):

- Left child of node i is at index $2i$.
- Right child of node i is at index $2i + 1$.

A node is a leaf if it has no children.

Derivation: For node i to have at least one child:

$$2i \leq n \Rightarrow i \leq \frac{n}{2}$$

Therefore:

- Nodes with indices 1 to $\left\lfloor \frac{n}{2} \right\rfloor$ are internal nodes.
- Nodes with indices $\left\lfloor \frac{n}{2} \right\rfloor + 1$ to n are leaf nodes.

Conclusion: All leaf nodes are located at indices $\left\lfloor \frac{n}{2} \right\rfloor + 1$ to n.

3 Heap Height Analysis and Build-Heap Complexity

3.1 (a) Number of Nodes at Height h

Claim 2. In a heap containing n elements, the number of nodes at height h is at most:

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

Explanation: Height is measured from the bottom of the heap. A node at height h must have a subtree containing at least 2^h nodes. Hence:

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

Tightening the bound gives:

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

3.2 (b) Time Complexity of Build-Heap

The Build-Heap algorithm applies Heapify to all non-leaf nodes. The cost of heapifying a node at height h is $O(h)$. The total cost is therefore:

$$\begin{aligned} T(n) &= \sum_{h=0}^{\log n} \left(\frac{n}{2^{h+1}} \cdot O(h) \right) \\ &= O(n) \sum_{h=0}^{\log n} \frac{h}{2^h} \end{aligned}$$

The series $\sum \frac{h}{2^h}$ converges to a constant.

Final Result:

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

Thus, the Build-Heap algorithm runs in linear time.

4 LU Decomposition Using Gaussian Elimination

Definition: LU decomposition factors a matrix A into the product:

$$A = LU$$

where:

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

Procedure:

1. Start with matrix A.
2. Use Gaussian elimination to eliminate elements below the diagonal.
3. Store the elimination multipliers in matrix L.
4. The resulting matrix after elimination forms U.

Example 1.

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

Eliminate a_{21} using the multiplier:

$$\ell_{21} = \frac{a_{21}}{a_{11}}$$

The resulting matrices are:

$$U = \begin{bmatrix} a_{11} & a_{12} \\ 0 & a_{22} - \ell_{21} a_{12} \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 \\ \ell_{21} & 1 \end{bmatrix}$$

5 Solving the LUP Recurrence Relation

Given:

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

Simplification:

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

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

Thus:

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

Final Computation:

$$\sum_{i=1}^{n} i = O(n^2) \quad \text{and} \quad \sum_{i=1}^{n} (n-i) = O(n^2)$$

Result:

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

6 Non-Singularity of the Schur Complement

Claim 3. Prove that if a matrix A is non-singular, then its Schur complement is also non-singular.

Explanation and Proof: Let matrix A be partitioned as:

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

where B is a square and non-singular matrix. The **Schur complement** of B in A is defined as:

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

We want to prove that if A is non-singular, then S is also non-singular.

Consider the block matrix factorization:

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

The determinant of A is:

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

Since A is non-singular:

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

and since B is non-singular:

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

It follows that:

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

Conclusion: The Schur complement S is non-singular.

7 Positive-Definite Matrices and LU Decomposition

Claim 4. Prove that positive-definite matrices are suitable for LU decomposition and do not require pivoting.

Explanation: A matrix A is **positive-definite** if:

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

Positive-definite matrices have the following properties:

- All leading principal minors are positive.
- All diagonal elements are non-zero.

Proof: During LU decomposition, division by pivot elements (diagonal entries of U) occurs. Pivoting is required only if a pivot is zero. For a positive-definite matrix:

$$\det(A_k) > 0$$

for all leading principal submatrices A_k. Thus, all pivots are strictly positive and non-zero.

Conclusion: LU decomposition can be performed without pivoting on positive-definite matrices, and no division by zero occurs.

8 BFS vs DFS for Finding Augmenting Paths

Question: For finding an augmenting path in a graph, should BFS or DFS be applied?

Answer: Breadth First Search (BFS) should be applied.

Justification: An **augmenting path** is a path that alternates between unmatched and matched edges and increases the size of a matching. BFS finds the shortest augmenting path in terms of number of edges. Using BFS:

- Guarantees shortest augmenting paths.
- Improves convergence speed.
- Ensures polynomial-time complexity (e.g., Hopcroft–Karp algorithm).

DFS may find longer paths, leading to inefficient augmentation.

Conclusion: BFS is preferred for finding augmenting paths.

9 Limitation of Dijkstra's Algorithm

Claim 5. Explain why Dijkstra's algorithm cannot be applied to graphs with negative edge weights.

Explanation: Dijkstra's algorithm assumes that once a vertex is selected with minimum tentative distance, that distance is final. This assumption fails in the presence of negative-weight edges.

Example: Suppose:

$$A \to B = 5, \quad B \to C = -10, \quad A \to C = 2$$

Dijkstra selects C first with distance 2. However, the path $A \to B \to C$ has total cost -5. Thus, the algorithm produces incorrect results.

Conclusion: Dijkstra's algorithm cannot handle negative weights because it relies on a greedy assumption that does not hold.

10 Symmetric Difference of Matchings

Claim 6. Prove that every connected component of the symmetric difference of two matchings is either a path or an even-length cycle.

Proof: Let M_1 and M_2 be two matchings in graph G. Define the symmetric difference:

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

Each vertex in $M_1 \oplus M_2$ has degree at most 2 because:

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

Thus, connected components must be:

- Paths (vertices of degree 1).
- Cycles (all vertices degree 2).

Since edges alternate between M_1 and M_2, cycles must be even-length.

Conclusion: Each component is either a path or an even-length cycle.

11 Definition of Co-NP

Definition: The complexity class Co-NP consists of all decision problems whose complements belong to NP. Formally:

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

Explanation: For a problem in Co-NP:

- A "NO" answer can be verified in polynomial time.
- A certificate exists for non-membership.

Example: The problem:

$$\text{TAUT} = \{\phi \mid \phi \text{ is true for all assignments}\}$$

is in Co-NP.

12 Verification of Boolean Circuit Output

Claim 7. Explain how the correctness of a Boolean circuit evaluating to true can be verified using DFS.

Explanation: A Boolean circuit is a directed acyclic graph (DAG) where:

- Nodes represent logic gates.
- Edges represent signal flow.

Verification Procedure:

1. Start DFS from the output gate.
2. Recursively visit all input gates.
3. Verify logical consistency at each gate.

Each gate is processed once.

Time Complexity:

$$O(V + E)$$

which is polynomial in circuit size.

Conclusion: Boolean circuit verification is in NP.

13 NP-Hardness of 3-SAT

Claim 8. Is the 3-SAT problem NP-Hard?

Answer: Yes, 3-SAT is NP-Hard.

Justification: Cook–Levin theorem shows that SAT is NP-Complete. Any SAT instance can be transformed into an equivalent 3-CNF formula in polynomial time. Thus:

$$\text{SAT} \leq_p \text{3-SAT}$$

Since 3-SAT is in NP and NP-Hard:

$$\text{3-SAT is NP-Complete}$$

14 Complexity of 2-SAT

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

Answer: 2-SAT is **not NP-Hard** and can be solved in polynomial time.

Explanation: 2-SAT clauses have at most two literals. The problem is reduced to implication graphs:

$$(a \lor b) \Rightarrow (\neg a \to b), (\neg b \to a)$$

A 2-SAT formula is satisfiable if no variable and its negation lie in the same strongly connected component.

Algorithm:

- Construct implication graph.
- Find SCCs using DFS or Kosaraju's algorithm.

Time Complexity:

$$O(V + E)$$

Conclusion: 2-SAT is solvable in polynomial time and belongs to class P.