Advanced Data Structures and Algorithms (ADSA)

Assignment

Name: Jyotirupa Basumatary
Roll No: A125007
Course / Semester: MTech CSE 1st semester

Question 1

Problem. Prove that the time complexity of the recursive `HEAPIFY` procedure is $\mathcal{O}(\log_2 n)$, given the recurrence

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

Solution:
We are given the recurrence

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

Let the constant amount of work performed outside the recursive call be $c > 0$. Then the recurrence can be rewritten as

$$T(n) = T\left(\frac{2n}{3}\right) + c.$$

Expanding the recurrence repeatedly, we obtain

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

The recursion terminates when the problem size becomes constant, say 1. Thus,

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

Rewriting the inequality,

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

Taking natural logarithms on both sides,

$$k \ln\left(\frac{2}{3}\right) \leq -\ln n.$$

Since $\ln(2/3) < 0$, dividing both sides by $\ln(2/3)$ reverses the inequality, yielding

$$k \geq \frac{\ln n}{\ln(3/2)}.$$

Hence, the number of recursive calls is

$$k = \Theta(\log n).$$

Because logarithms with different bases differ only by a constant factor,

$$\Theta(\log n) = \Theta(\log_2 n).$$

At the base case,

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

Substituting this into the expanded recurrence gives

$$T(n) = \mathcal{O}(1) + kc.$$

Since $k = \Theta(\log n)$, it follows that

$$\boxed{T(n) = \mathcal{O}(\log_2 n)}.$$

Question 2

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

Solution:
A binary heap is a complete binary tree stored in an array. For a node stored at index i, the indices of its children are given by

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

A node is a leaf node if it has no children. Thus, a node at index i is a leaf if

$$2i > n.$$

Solving for i, we get

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

Since array indices are integers, all nodes with indices

$$i \geq \left\lfloor \frac{n}{2} \right\rfloor + 1$$

are leaf nodes.

The maximum index in the heap is n. Hence, all leaf nodes are located at indices from

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

$$\boxed{\text{Hence proved.}}$$

Question 3

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

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

Solution:
A binary heap is a complete binary tree. The height of a node is defined as the number of edges on the longest downward path from that node to a leaf.

In a complete binary tree, the number of nodes decreases exponentially as the height increases. Specifically:

- At height 0 (leaf nodes), there are at most $\frac{n}{2}$ nodes,
- At height 1, there are at most $\frac{n}{4}$ nodes,
- At height 2, there are at most $\frac{n}{8}$ nodes.

Continuing in this manner, at height h, the number of nodes is at most

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

Hence, the number of nodes at height h is at most

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

Problem(b) Using the above result, prove that the time complexity of the `BUILD-HEAP` algorithm is $O(n)$.

Solution:
The `BUILD-HEAP` algorithm constructs a heap by calling `HEAPIFY` on each non-leaf node, starting from the lowest level and moving up to the root.

The time complexity of `HEAPIFY` depends on the height of the node. If a node is at height h, the time required by `HEAPIFY` is $O(h)$.

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

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

Therefore, the total time complexity of the `BUILD-HEAP` algorithm can be expressed as

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

Taking n outside the summation, we get

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

The series

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

converges to a constant.

Hence,

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

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

Question 4

Problem. Explain the LU decomposition of a matrix using Gaussian Elimination. Clearly describe each step involved in the process and illustrate it with an example.

Solution:

LU decomposition factors a square matrix A into the product of a lower triangular matrix L and an upper triangular matrix U, such that

$$A = LU.$$

This factorization is obtained using Gaussian Elimination.

Step 1: Start with the matrix A

Consider a square matrix

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}.$$

Step 2: Apply Gaussian Elimination

Using elementary row operations, eliminate the entries below the main diagonal to convert A into an upper triangular matrix U.

For each column k, the multiplier used to eliminate the entry a_{ik} is

$$m_{ik} = \frac{a_{ik}}{a_{kk}}, \quad i > k.$$

Step 3: Construct the matrix L

The multipliers m_{ik} used during elimination are stored in the lower triangular matrix L, while the diagonal entries of L are set to 1:

$$L = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ m_{21} & 1 & 0 & \cdots & 0 \\ m_{31} & m_{32} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ m_{n1} & m_{n2} & m_{n3} & \cdots & 1 \end{bmatrix}.$$

Step 4: Obtain the matrix U

After completing Gaussian elimination, the resulting upper triangular matrix is denoted by U:

$$U = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix}.$$

Step 5: Final decomposition

Thus, the original matrix A can be expressed as

$$\boxed{A = LU}.$$

Conclusion

LU decomposition uses Gaussian elimination to transform a matrix into an upper triangular form while recording the elimination multipliers in a lower triangular matrix. This decomposition is useful for efficiently solving systems of linear equations. **Example.**

Example: LU Decomposition

Let

$$A = \begin{bmatrix} 2 & 3 & 1 \\ 4 & 7 & 7 \\ -2 & 4 & 5 \end{bmatrix}.$$

Step 1: Eliminate entries below a_{11}

Pivot element: $a_{11} = 2$

Multipliers:

$$m_{21} = \frac{4}{2} = 2, \qquad m_{31} = \frac{-2}{2} = -1.$$

Row operations:

$$R_2 \leftarrow R_2 - 2R_1, \qquad R_3 \leftarrow R_3 + R_1.$$

This gives

$$\begin{bmatrix} 2 & 3 & 1 \\ 0 & 1 & 5 \\ 0 & 7 & 6 \end{bmatrix}.$$

Step 2: Eliminate entry below a_{22}

Pivot element: $a_{22} = 1$

Multiplier:

$$m_{32} = \frac{7}{1} = 7.$$

Row operation:

$$R_3 \leftarrow R_3 - 7R_2.$$

This gives the upper triangular matrix

$$U = \begin{bmatrix} 2 & 3 & 1 \\ 0 & 1 & 5 \\ 0 & 0 & -29 \end{bmatrix}.$$

Step 3: Construct the lower triangular matrix

The multipliers used are placed in L, with diagonal entries equal to 1:

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 7 & 1 \end{bmatrix}.$$

Step 4: Verify that $A = LU$

$$LU = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 7 & 1 \end{bmatrix} \begin{bmatrix} 2 & 3 & 1 \\ 0 & 1 & 5 \\ 0 & 0 & -29 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 \\ 4 & 7 & 7 \\ -2 & 4 & 5 \end{bmatrix}.$$

Hence,

$$\boxed{A = LU}.$$

Question 5

Problem Solve the following recurrence relation arising from the LUP decomposition 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:

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

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

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

$$= \mathrm{O(n)} + \mathrm{O}\left(\tfrac{n(n+1)}{2} - n\right) + O(n) + O\left(n^2 - \tfrac{n(n+1)}{2}\right)$$

$$= \mathrm{O(n)} + \mathrm{O}\left(\tfrac{n^2-n}{2}\right) + O(n) + O\left(\tfrac{n^2-n}{2}\right)$$

$$= \mathrm{O(n^2)} + O(n) + O(n^2) + O(n)$$

$$= \mathrm{O(n^2)}.$$

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

Question 6

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

Solution:

Let the matrix A be partitioned as

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

where A_{11} is a square and non-singular matrix.

The Schur complement of A_{11} in A is defined as

$$S = A_{22} - A_{21}A_{11}^{-1}A_{12}.$$

Since A_{11} is non-singular, the matrix A can be factorized as

$$A = \begin{bmatrix} I & 0 \\ A_{21}A_{11}^{-1} & I \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} \\ 0 & S \end{bmatrix}.$$

Both matrices on the right-hand side are block triangular matrices. Therefore, the determinant of A is given by

$$\det(A) = \det(A_{11}) \cdot \det(S).$$

Since A is non-singular, we have

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

Also, because A_{11} is non-singular,

$$\det(A_{11}) \neq 0.$$

Hence, it follows that

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

Therefore, the Schur complement S is non-singular.

Hence proved.

Question 7

Problem 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:
In LU decomposition using Gaussian elimination, the matrix is reduced recursively by eliminating entries below the diagonal. At each stage of this process, division is performed by a pivot element, which is the current diagonal entry. Pivoting is required only when a pivot element becomes zero, since division by zero must be avoided.

Let A be a positive-definite matrix. For such a matrix, all leading principal submatrices are non-singular. In particular, if A_k denotes the leading $k \times k$ submatrix of A, then

$$\det(A_k) > 0 \quad \text{for all } k.$$

In the LU decomposition process, the pivot used at the k-th stage corresponds to the diagonal element associated with the leading principal submatrix A_k. Since $\det(A_k) > 0$, the corresponding pivot element is non-zero.

Because this holds at every stage of the recursive elimination, division by zero never occurs during LU decomposition of a positive-definite matrix. Therefore, LU decomposition can be carried out without pivoting.

Hence, positive-definite matrices are suitable for LU decomposition and do not require pivoting to avoid division by zero in the recursive strategy.

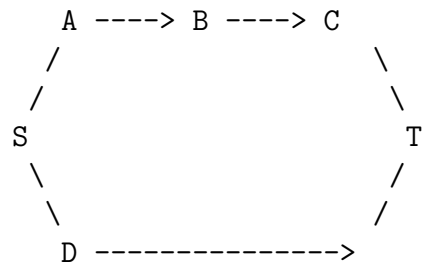
Question 8

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

Solution:
For finding an augmenting path in a graph, Breadth First Search (BFS) is preferred over Depth First Search (DFS).

An augmenting path is a path from the source S to the sink T along which the flow can be increased. Although both BFS and DFS can be used to find such a path, the choice of search strategy affects the efficiency of the algorithm.

Example.
Consider the following flow network:



There are two possible paths from the source S to the sink T:

$$S \to A \to B \to C \to T \quad \text{(long path)}$$

$$S \to D \to T \quad \text{(short path)}$$

Assume all edges have available capacity.

Using DFS
Depth First Search starts from S and explores one path as deeply as possible before considering other paths. Hence, DFS may first find the long path

$$S \to A \to B \to C \to T.$$

This path has more edges, so augmenting flow along it is inefficient. DFS may repeatedly choose such long paths, increasing the number of augmentations and reducing efficiency.

Using BFS
Breadth First Search explores the graph level by level. From S, BFS first examines nodes one edge away and quickly finds the shorter path

$$S \to D \to T.$$

This is the shortest augmenting path in terms of the number of edges. Using shorter paths reduces the total number of augmentations and improves the efficiency of the algorithm. This idea is used in algorithms such as Edmonds–Karp.

Conclusion

DFS may find long and inefficient augmenting paths, whereas BFS always finds the shortest augmenting path first. Hence, BFS is preferred over DFS for finding augmenting paths in a graph.

Question 9

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

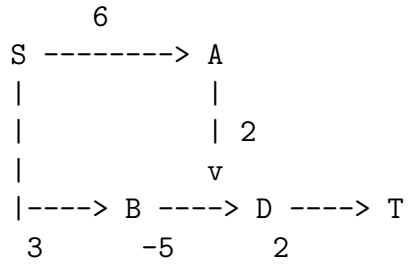
Solution:

Dijkstra's algorithm constructs shortest paths by repeatedly selecting the vertex with the smallest temporary distance and permanently fixing that distance. Once a vertex is selected, the algorithm assumes that its shortest distance cannot be improved later.

This assumption holds only when all edge weights are non-negative. With non-negative weights, any additional path to a vertex can only increase the total distance. However, if negative edge weights are present, a shorter path to a vertex may be discovered later through a negative edge. Since Dijkstra's algorithm never revisits a vertex whose distance has been finalized, it may produce incorrect results.

Explanation using an example

Consider the following directed weighted graph:



The edges in the graph are:

$$S \to A = 6, \quad S \to B = 3, \quad B \to A = -5, \quad A \to D = 2, \quad B \to D = 4, \quad D \to T = 2.$$

From the source S to node A, there are two possible paths:

$$S \to A = 6$$

and

$$S \to B \to A = 3 + (-5) = -2.$$

Thus, the actual shortest distance to A is -2.

During the execution of Dijkstra's algorithm, node A may be finalized with distance 6 before the algorithm discovers the shorter path through B. Once A is finalized, the algorithm does not update its distance even after finding the shorter path using the negative edge.

Hence, the algorithm reports an incorrect shortest distance.

Conclusion

Negative edge weights can reduce the distance to a vertex after it has already been finalized. This violates the fundamental assumption of Dijkstra's algorithm. Therefore, Dijkstra's algorithm cannot be applied to graphs with negative edge weights.

Question 10

Problem 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 a graph G. The symmetric difference of M_1 and M_2 is defined as

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

Since M_1 and M_2 are matchings, each vertex in G is incident to at most one edge of M_1 and at most one edge of M_2. Hence, in the graph formed by the edges of $M_1 \oplus M_2$, every vertex has degree at most two.

As a result, each connected component of $M_1 \oplus M_2$ is either a path or a cycle. If a component contains a vertex of degree one, it is a path. If all vertices in a component have degree two, it forms a cycle.

Moreover, along any such path or cycle, the edges must alternate between M_1 and M_2, since no two edges from the same matching can be adjacent. In the case of a cycle, this alternation implies that the number of edges must be even.

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

Question 11

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

Solution:

The complexity class Co-NP consists of all decision problems for which a NO answer can be verified in polynomial time.

In other words, a problem is in Co-NP if, whenever the correct answer is NO, there exists a certificate (proof) that can be checked efficiently (in polynomial time).

Another equivalent way to define Co-NP is:

A problem belongs to Co-NP if its complement problem belongs to NP.

Type of problems in Co-NP

Problems in Co-NP are typically those where it is easy to verify that a given statement is false, even if it may be hard to verify that it is true.

To understand this, compare NP and Co-NP:

- In **NP**, we can efficiently verify **YES-instances** using a certificate.
- In **Co-NP**, we can efficiently verify **NO-instances** using a certificate.

Many Co-NP problems arise as the complements of NP problems.

Examples

1. **UNSAT (Unsatisfiability)**
   - SAT (Boolean satisfiability) is an NP problem.
   - UNSAT asks whether a Boolean formula has no satisfying assignment.
   - A proof that no assignment satisfies the formula verifies a NO-instance.
   - Hence, UNSAT belongs to Co-NP.
2. **Tautology problem**
   - The problem asks whether a Boolean formula is true for all possible assignments.
   - To show that a formula is not a tautology, it is enough to give one assignment where it is false.
   - Therefore, the tautology problem belongs to Co-NP.

Key idea (simple words)

- **NP**: easy to verify that an answer is **YES**
- **Co-NP**: easy to verify that an answer is **NO**

Whether **NP = Co-NP** is one of the major open questions in theoretical computer science.

Question 12

Problem 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 input variables are leaf nodes, logic gates (AND, OR, NOT) are internal nodes, and the output gate is the root of the graph.

Given that the output of the Boolean circuit evaluates to true, its correctness can be verified using Depth First Search (DFS). DFS starts from the output gate and recursively visits all gates and input variables that contribute to the output.

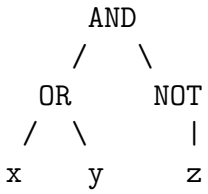
During DFS, each gate is verified according to its logical operation:

- An **AND gate** is correct if all of its input gates evaluate to true.
- An **OR gate** is correct if at least one of its input gates evaluates to true.
- A **NOT gate** is correct if its output is the negation of its input value.

DFS continues until the input variables are reached. Since the values of the input variables are given, their correctness can be directly verified. The computed values are then propagated back through the circuit to confirm that each gate produces the correct output, finally verifying the output gate.

Example

Consider the following Boolean circuit:



The circuit represents the Boolean expression

$$(x \vee y) \wedge (\neg z).$$

Let the input values be:

$$x = 1, \quad y = 0, \quad z = 0.$$

Using DFS, the OR gate is verified first. Since $x = 1$, the OR gate evaluates to true. The NOT gate is then verified. Since $z = 0$, the NOT gate evaluates to true. As both inputs to the AND gate are true, the AND gate evaluates to true, confirming the correctness of the circuit output.

Time Complexity

Each gate and each wire in the circuit is visited at most once during DFS, and each gate evaluation takes constant time. Therefore, if the circuit contains n gates and connections, the verification takes $\mathcal{O}(n)$ time, which is polynomial.

Conclusion

By representing the Boolean circuit as a directed acyclic graph and performing a Depth First Search starting from the output gate, the correctness of a Boolean circuit whose output evaluates to true can be verified in polynomial time.

Question 13

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

Solution:

Yes, the 3-SAT (3-CNF-SAT) problem is NP-Hard.

The Boolean Satisfiability problem (SAT) is a well-known NP-Complete problem. This means that every problem in NP can be reduced to SAT in polynomial time.

It has been shown that any SAT instance can be transformed into an equivalent 3-SAT instance in polynomial time. This transformation converts a Boolean formula into a conjunctive normal form where each clause contains exactly three literals, without changing the satisfiability of the formula.

Since SAT can be reduced to 3-SAT in polynomial time and SAT is NP-Hard, it follows that 3-SAT is also NP-Hard.

Moreover, 3-SAT belongs to NP because a given truth assignment can be verified in polynomial time. Therefore, 3-SAT is not only NP-Hard but also NP-Complete.

Since SAT reduces to 3-SAT in polynomial time, the 3-SAT problem is NP-Hard (and in fact NP-Complete).

Question 14

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

Solution:

The 2-SAT problem is not NP-Hard, and it can be solved in polynomial time.

The 2-SAT problem is a restricted version of the Boolean satisfiability problem in which each clause contains at most two literals. Unlike SAT and 3-SAT, which are NP-Complete, this restriction makes the problem easier to solve.

Each clause of the form

$$(a \vee b)$$

can be rewritten as two implications:

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

Using these implications, a directed implication graph can be constructed. A 2-SAT instance is satisfiable if and only if no variable and its negation belong to the same strongly connected component of the graph.

Strongly connected components can be computed using algorithms such as Kosaraju's or Tarjan's algorithm, which run in linear time. Hence, the overall time complexity of solving 2-SAT is polynomial.

Since 2-SAT can be solved in polynomial time, it is not NP-Hard (unless $P = NP$).

Conclusion

The 2-SAT problem is not NP-Hard and admits a polynomial-time solution.