ADSA_ASSIGNMENT

November 26, 2025

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 done outside the recursive call be $c > 0$. Then we can rewrite the recurrence as

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

Step 1: Expand the recurrence.
Start from

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

Apply the recurrence to the subproblem $T(2n/3)$:

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

Substitute this back:

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

Apply the recurrence once more:

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

so

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

Continuing in this way, after k steps we obtain the general form

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

Step 2: Recursion stops

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

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

Rewriting,

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

Take natural logarithms on both sides:

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

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

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

Hence the number of levels of recursion is

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

where the base of the logarithm is a constant (here effectively base $3/2$). Changing the base of a logarithm only introduces a constant factor, so

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

Step 3: Substitute back and conclude.

At the base of the recursion, when the argument is constant, we have

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

Using the expanded form

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

we obtain

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

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

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

In particular, up to a constant factor, this is $\mathcal{O}(\log_2 n)$.

Therefore, The time complexity of the recursive `HEAPIFY` procedure is

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

Question 2

Problem. In an array of size n representing a binary heap (with 1-based indexing), the leaf nodes are exactly at indices

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

Solution.

We assume a binary heap stored in an array $A[1 \ldots n]$ using the standard indexing rules:

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

A node is a leaf if it has no children within the array, i.e., if both of its child indices are greater than n.

(1) Nodes with index $i > \lfloor \frac{n}{2} \rfloor$ are leaves.

Let i be an index such that

$$i > \left\lfloor \frac{n}{2} \right\rfloor.$$

Consider its left child index $2i$. Since $i > \frac{n}{2}$, we have

$$2i > n.$$

Thus the left child of i does not exist in the array. The right child index is $2i+1$, and

$$2i+1 > 2i > n,$$

so the right child also does not exist. Therefore, node i has no children and is a leaf.

Hence all indices

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

correspond to leaf nodes.

(2) Nodes with index $1 \le i \le \lfloor \frac{n}{2} \rfloor$ are internal nodes.

Now let i be such that

$$1 \le i \le \left\lfloor \frac{n}{2} \right\rfloor.$$

Its left child is at index $2i$. Since $i \le \frac{n}{2}$, it follows that

$$2i \le n.$$

Thus the left child index $2i$ is within the array bounds, so node i has at least one child and is therefore not a leaf.

From (1) and (2), we conclude that:

- The internal nodes are exactly at indices $1, 2, \ldots, \left\lfloor \frac{n}{2} \right\rfloor$.
- The leaf nodes are exactly at indices $\left\lfloor \frac{n}{2} \right\rfloor + 1, \ldots, n$.

In an array representation of a binary heap of size n, the leaf nodes start from index

$$\boxed{\left\lfloor \frac{n}{2} \right\rfloor + 1, 2, 3, 4\ldots\ldots\ldots, n.}$$

Question 3

Problem: 3.a) Show that there at atmost $ceil[n/2^(n+1)]$ nodes at height h in any n element heap.
b) prove TC¡= O(n)

(a) Number of nodes at height h

Claim. In an n-element heap, the number of nodes at height h is at most

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

Proof. View the heap as a complete binary tree. Consider any node at height h (height counted from the bottom, so leaves have height 0). The subtree

rooted at this node is itself a complete binary tree of height h, and the smallest complete binary tree of height h has

$$1 + 2 + 4 + \cdots + 2^h = 2^{h+1} - 1$$

nodes.

Thus each node at height h is the root of a subtree containing at least $2^{h+1}-1$ nodes. Let there be k nodes at height h. Their subtrees are pairwise disjoint and all are contained in the heap of size n, so

$$n \ \geq \ k(2^{h+1} - 1).$$

Hence

$$k \leq \frac{n}{2^{h+1} - 1} < \frac{n}{2^{h+1}}.$$

Therefore, the number of nodes at height h satisfies

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

(b) Time complexity $TC \leq O(n)$

The standard `BUILD-HEAP` algorithm is:

```
for i = floor(n/2) downto 1:
    HEAPIFY(A, i)
```

The running time of `HEAPIFY` on a node is proportional to its height. Let h be the height of a node. From part (a), the number of nodes at height h is at most $\frac{n}{2^{h+1}}$.

Thus the total time $T(n)$ to build the heap is bounded by

$$T(n) \leq \sum_{h=0}^{H} \left(\frac{n}{2^{h+1}} \right) \cdot ch = cn \sum_{h=0}^{H} \frac{h}{2^{h+1}},$$

for some constant $c > 0$, where $H = O(\log n)$ is the maximum height of the heap.

The infinite series $\sum_{h=0}^{\infty} \frac{h}{2^h}$ converges to a finite constant, so the partial sum is $O(1)$. Therefore,

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

Conclusion: The time complexity of building an n-element heap satisfies

$$\boxed{T(n) \leq O(n)}.$$

Question 4

Problem: LU decomposition of a matrix using Gaussian elimination

Solution: The LU decomposition of a matrix A is a factorization

$$A = LU,$$

where L is a lower triangular matrix with 1's on the diagonal and U is an upper triangular matrix. The decomposition can be obtained directly from Gaussian elimination by storing the multipliers used in the elimination steps into the matrix L.

Example

Consider the matrix

$$A = \begin{bmatrix} 2 & 3 & 1 \\ 4 & 7 & 7 \\ 6 & 18 & 22 \end{bmatrix}.$$

Step 1: Eliminate entries below a_{11}

Compute multipliers:

$$\ell_{21} = \frac{4}{2} = 2, \qquad \ell_{31} = \frac{6}{2} = 3.$$

Update rows:

$$R_2 \leftarrow R_2 - 2R_1 = [0\ 1\ 5],$$
$$R_3 \leftarrow R_3 - 3R_1 = [0\ 9\ 19].$$

Step 2: Eliminate below a_{22}

Compute multiplier:

$$\ell_{32} = \frac{9}{1} = 9.$$

Update row:

$$R_3 \leftarrow R_3 - 9R_2 = [0\ 0\ -26].$$

Result

Thus,

$$U = \begin{bmatrix} 2 & 3 & 1 \\ 0 & 1 & 5 \\ 0 & 0 & -26 \end{bmatrix}, \qquad L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 9 & 1 \end{bmatrix}.$$

$$A = LU$$

which completes the LU decomposition.

Question 5

Problem: Solve recurrence relation

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

for LUP solve procedure.

Solution.

We interpret each $O(1)$ term as a constant $c > 0$. Then

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

First sum:

$$S_1 = \sum_{i=1}^{n} \Big(c + \sum_{j=1}^{i-1} c\Big) = \sum_{i=1}^{n} \big(c + c(i-1)\big) = \sum_{i=1}^{n} ci = c \sum_{i=1}^{n} i = c \cdot \frac{n(n+1)}{2} = \Theta(n^2).$$

Second sum:

$$S_2 = \sum_{i=1}^{n} \Big(c + \sum_{j=i+1}^{n} c\Big) = \sum_{i=1}^{n} \big(c + c(n-i)\big) = \sum_{i=1}^{n} c(n-i+1).$$

As i goes from 1 to n, the values $(n - i + 1)$ run through $1, 2, \ldots, n$, so

$$S_2 = c \sum_{k=1}^{n} k = c \cdot \frac{n(n+1)}{2} = \Theta(n^2).$$

Therefore,

$$T(n) = S_1 + S_2 = \Theta(n^2) + \Theta(n^2) = \Theta(n^2).$$

Final answer:

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

Question 5

Problem: Solve recurrence relation

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

for LUP solve procedure.

Solution.

We interpret each $O(1)$ term as a constant $c > 0$. Then

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

First sum:

$$S_1 = \sum_{i=1}^{n} \Big(c + \sum_{j=1}^{i-1} c\Big) = \sum_{i=1}^{n} \big(c + c(i-1)\big) = \sum_{i=1}^{n} ci = c \sum_{i=1}^{n} i = c \cdot \frac{n(n+1)}{2} = \Theta(n^2).$$

Second sum:

$$S_2 = \sum_{i=1}^{n} \Big(c + \sum_{j=i+1}^{n} c\Big) = \sum_{i=1}^{n} \big(c + c(n-i)\big) = \sum_{i=1}^{n} c(n-i+1).$$

As i goes from 1 to n, the values $(n - i + 1)$ run through $1, 2, \ldots, n$, so

$$S_2 = c \sum_{k=1}^{n} k = c \cdot \frac{n(n+1)}{2} = \Theta(n^2).$$

Therefore,

$$T(n) = S_1 + S_2 = \Theta(n^2) + \Theta(n^2) = \Theta(n^2).$$

Final answer:

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

Question 6

Problem: Prove that if A is non singular, then the schur complement of A is also non-singular.

Clarification. Let

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

where A is a square, nonsingular matrix. The Schur complement of A in M is defined as

$$S = D - CA^{-1}B.$$

In general, the assumption "A is nonsingular" alone is not sufficient to guarantee that S is nonsingular. The standard result states:

If M and A are both nonsingular, then the Schur complement

$$S = D - CA^{-1}B$$

is also nonsingular.

We prove this statement.

Proof. Assume that M and A are nonsingular, and let

$$S = D - CA^{-1}B.$$

Suppose, for contradiction, that S is singular. Then there exists a nonzero vector $x \neq 0$ such that

$$Sx = (D - CA^{-1}B)x = 0.$$

Define

$$y = -A^{-1}Bx.$$

Then

$$Ay + Bx = A(-A^{-1}Bx) + Bx = -Bx + Bx = 0.$$

Consider the block vector

$$z = \begin{bmatrix} y \\ x \end{bmatrix} \neq 0.$$

We compute

$$Mz = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} y \\ x \end{bmatrix} = \begin{bmatrix} Ay + Bx \\ Cy + Dx \end{bmatrix}.$$

We already have $Ay + Bx = 0$. For the second block,

$$Cy + Dx = C(-A^{-1}Bx) + Dx = -CA^{-1}Bx + Dx = (D - CA^{-1}B)x = Sx = 0.$$

Hence

$$Mz = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

so z is a nonzero vector in the nullspace of M. This implies that M is singular, contradicting the assumption that M is nonsingular.

Therefore our assumption that S is singular must be false. Hence the Schur complement

$$S = D - CA^{-1}B$$

is nonsingular.

Question 7

Problem: Prove that positive-definite matrices work well with LU decomposition and don't require any pivoting to avoid divide by zero in the recursive strategy.

Solution.

Let $A \in R^{n\times n}$ be a real symmetric positive-definite (SPD) matrix. By definition,

$$A = A^T \quad \text{and} \quad x^T Ax > 0 \quad \text{for all } x \neq 0.$$

Denote by A_k the leading principal submatrix of A of order k, i.e., the top-left $k \times k$ block of A.

Claim: For each $k = 1, 2, \ldots, n$, the matrix A_k is itself symmetric positive-definite and hence nonsingular.

Proof of claim. Let $y \in R^k$ be any nonzero vector, and extend it to a vector $x \in R^n$ by padding with zeros in the remaining $(n - k)$ entries:

$$x = \begin{bmatrix} y \\ 0 \end{bmatrix}.$$

Then

$$x^T Ax = y^T A_k y.$$

Since A is positive-definite, $x^T Ax > 0$ for all $x \neq 0$. Thus for all $y \neq 0$ we have $y^T A_k y > 0$, so A_k is positive-definite. In particular, each A_k is nonsingular and $\det(A_k) > 0$.

A standard result from LU factorization theory states that a matrix A admits an LU decomposition without pivoting, i.e.,

$$A = LU,$$

with L unit lower triangular and U upper triangular, if and only if all its leading principal submatrices A_k are nonsingular.

Since for an SPD matrix A every leading principal submatrix A_k is nonsingular (as shown above), it follows that A has an LU factorization without any row exchanges or pivoting.

Furthermore, during Gaussian elimination (or a recursive LU strategy), the pivots are precisely the diagonal elements corresponding to these leading principal submatrices. Because each A_k is nonsingular, none of these pivots is zero. Hence no division by zero occurs in the elimination process, and no pivoting is needed to avoid such situations.

Conclusion: Positive-definite matrices work well with LU decomposition: all leading principal minors are positive, all pivots are nonzero, and LU (or Cholesky) factorization can be performed without pivoting, with no risk of division by zero in the recursive strategy.

Question 8

Problem: For augmenting path should we apply BFS or DFS?

Solution In the search for augmenting paths in flow networks or bipartite matching problems, BFS is preferred over DFS. Using BFS ensures that we always find the shortest augmenting path in terms of number of edges. This is the foundation of the Edmonds–Karp improvement of the Ford–Fulkerson method, which guarantees a polynomial running time by ensuring that each augmenting path increases in length in a controlled manner. DFS, on the other hand, may find long or inefficient augmenting paths, causing excessive backtracking and possibly leading to exponential running time in the worst case. Therefore, BFS is the appropriate choice when searching for augmenting paths because of its superior efficiency and guaranteed performance.

Question 9

Problem: Why Dijkstra cannot be applied to negative weights?

Solution Dijkstra's algorithm relies on the assumption that once the shortest distance to a vertex has been determined and the vertex is "finalized," that distance will never need to be improved. This property holds only when all edge weights are non-negative. If a negative-weight edge exists, a path that initially appears longer may later become shorter by using the negative-weight edge, meaning that a supposedly finalized distance could actually be incorrect. Since Dijkstra's algorithm never revisits or updates a finalized vertex, it fails to compute the correct shortest paths in such cases. Therefore, Dijkstra's algorithm cannot be used when negative weights are present, and algorithms such as Bellman–Ford are required instead.

Question 10

Problem: Every connected component of the symmetric difference of two matchings in a graph G is a path or an even cycle.

Solution Let M_1 and M_2 be two matchings in a graph G, and consider their symmetric difference

$$M_1 \,\Delta\, M_2 = (M_1 \setminus M_2) \cup (M_2 \setminus M_1).$$

In a matching, each vertex is incident to at most one edge. Therefore, in the symmetric difference $M_1 \Delta M_2$, each vertex may be incident to at most one edge from M_1 and at most one edge from M_2. Hence every vertex in $M_1 \Delta M_2$ has degree at most 2.

Any graph in which every vertex has degree at most 2 must consist of connected components that are either paths or cycles. Furthermore, along each

component, the edges must alternate between M_1 and M_2, since no two consecutive edges can belong to the same matching. In a cycle, this alternating pattern forces the cycle to have even length. In a path, the endpoints have degree 1, and the internal vertices have degree 2, again with edges alternating between M_1 and M_2.

Thus, every connected component of the symmetric difference $M_1 \Delta M_2$ is either an alternating path or an alternating even cycle, as required.

Question 11

Problem: What are Co-NP class problems?

Solution: Co-NP is the class of decision problems that are the complements of NP problems. Formally, a language L is in Co-NP if and only if its complement $\overline{L}$ is in NP. In NP, every "yes" instance of a problem has a polynomial-size certificate that can be verified in polynomial time. In contrast, in Co-NP, every "no" instance has a polynomial-size certificate that can be verified efficiently.

An important example is the problem of determining whether a Boolean formula is *unsatisfiable*. While a satisfying assignment is a short certificate for a "yes" answer in SAT (an NP problem), a certificate for unsatisfiability can be verified in polynomial time, making UNSAT a Co-NP problem. It is unknown whether NP equals Co-NP, although most complexity theorists believe the two classes are distinct. Thus, Co-NP consists of problems whose "no" answers can be checked efficiently.