1 Solution to Problem 1

1.1 Probability of Picking a "Good" Pivot

Because the pivot is chosen uniformly at random among n elements, the probability of picking such a pivot in a single try is

$$\frac{n/3}{n} = \frac{1}{3}.$$

Hence, the expected number of tries until we get a good pivot is

$$\mathbb{E}[\text{tries}] = \frac{1}{(1/3)} = 3.$$

1.2 Cost of Partitioning

Partitioning an array of size n takes $O(n)$ time. Since we expect to make 3 attempts per pivot selection (on average) before succeeding, the expected time to pick and partition around a good pivot is $3 \times O(n) = O(n)$. Although the constant factor 3 might be relevant in a more detailed analysis, it does not affect the asymptotic form.

1.3 Recurrence for the Expected Running Time

Once a good pivot is found, the array of size n is split into two subproblems, each of size at most $\frac{2}{3}n$. If $T(n)$ denotes the expected running time to sort n elements:

- We pay $O(n)$ for the (expected) pivot selection and partition.
- We then solve **two** subproblems of size at most $\frac{2n}{3}$.

Thus, the recurrence relation is:

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

1.4 Solving the Recurrence

Suppose $T(n)$ behaves like $c\,n^p$. Then:

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

Plugging into the recurrence:

$$c\,n^p \approx O(n) + 2\left[c\frac{2^p}{3^p}n^p\right].$$

For large n, the linear term $O(n)$ is negligible compared to n^p if $p > 1$. Thus, we focus on:

$$c\,n^p \approx 2\,c\,\frac{2^p}{3^p}n^p.$$

Canceling $c\,n^p$ (assuming $c \neq 0$ and $n^p \neq 0$) gives:

$$1 = 2 \times \frac{2^p}{3^p} \quad \Rightarrow \quad \frac{2^p}{3^p} = \frac{1}{2} \quad \Rightarrow \quad \left(\frac{2}{3}\right)^p = \frac{1}{2} \quad \Rightarrow \quad \left(\frac{3}{2}\right)^p = 2.$$

Taking logs (in any base) gives:

$$p = \log_{\frac{3}{2}}(2) = \frac{\ln(2)}{\ln\left(\frac{3}{2}\right)} \approx 1.7095.$$

Hence,

$$T(n) = \Theta\left(n^{\log_{3/2}(2)}\right) \approx \Theta\big(n^{1.7095}\big).$$

1.5 Conclusion

Because we insist on picking pivots that split the array so each side has at most $\frac{2}{3}$ of the elements, we get a recurrence of the form

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

whose solution grows as n^p with $p \approx 1.7095$. Therefore, the **expected running time** of this variant of Quicksort is

$$\boxed{\Theta\big(n^{1.7095}\big)}.$$

2 Solution to Problem 2

2.1 Standard RBT Insertion Complexity

Inserting a node into a red-black tree involves:

1. Performing a standard binary-search-tree (BST) insertion, which takes $O(\log n)$ time because the tree's height is $O(\log n)$.
2. Potentially re-coloring and performing up to a constant number of rotations. Each rotation or re-coloring step is $O(1)$, and there can be at most $O(\log n)$ such steps along the path from the newly inserted node to the root.

Hence, **without** any augmentation, the worst-case insertion time in a red-black tree is $O(\log n)$.

2.2 Augmenting with Black-Height

We define the *black-height* of a node in a red-black tree as:

> The number of black nodes on any path from that node down to a leaf.

If we store this black-height as an additional integer field in each node, we must update it whenever a rotation or re-coloring affects the node's subtree.

2.3 Why the Augmentation Does *Not* Increase Complexity

Local Updates

- When we insert a node and then fix any violations, we only modify a small, *constant-sized* portion of the tree at each step. For instance, a rotation involves only a node, its parent, and possibly its grandparent.
- Because the black-height of a node can be computed using the black-heights of its children plus one (if the node itself is black), adjusting the black-heights during a rotation or recoloring is an $O(1)$ operation. It only requires looking at a few pointers and performing simple arithmetic.

At Most $O(\log n)$ Fix-Up Steps

- The insertion fix-up loop in a red-black tree can move upward from the inserted node to the root, but in the worst case it will pass through $O(\log n)$ levels (because the height is $O(\log n)$).
- Each step of the fix-up is still $O(1)$ even with black-height updates. Hence, the total overhead of insertion plus fix-up remains $O(\log n)$.

No Global Recomputation

- We do *not* need to recompute black-heights for *all* nodes after each insertion; only the nodes along the path of insertion or those involved in a rotation need updating. This again ensures the total work per insertion remains proportional to $\log n$.

Consequently, **maintaining black-height in each node adds only a constant amount of work to each fix-up step**, so the overall insertion complexity remains $O(\log n)$ in the worst case.

2.4 Conclusion

Augmenting each node with a black-height field allows us to query any node's black-height in $O(1)$ time without increasing the worst-case insertion complexity. The fix-up procedure still performs at most $O(\log n)$ steps, and each step—including updating black-height—costs constant time. Therefore, the worst-case running time for inserting into a red-black tree **remains** $O(\log n)$.

3 Solution to Problem 3

3.1 Standard RBT Insertion Complexity (No Depth Field)

- **Insertion Steps:**
  1. Insert the new node using a regular BST insertion ($O(\log n)$ time).
  2. Fix any violations of RBT properties by recoloring or performing a **constant number** of rotations at each step. This fix-up path can be at most $O(\log n)$ levels from the inserted node to the root.
- **Total Time:** Each step of recoloring or rotation is $O(1)$, repeated $O(\log n)$ times, giving $O(\log n)$ worst-case insertion.

3.2 Augmenting Each Node with a Depth Field

We now add a field `depth[u]` to each node u. By definition:

$$\text{depth}[u] = 1 + \text{depth}[\text{parent}(u)],$$

where the root's depth is typically 1 (or 0, depending on convention).

Key Point: If a node's parent changes or if the parent's depth changes, then the node's depth must also be updated. This propagates down to all descendants in that subtree.

3.3 Why Maintaining Depth Can Increase Insertion Time

When you insert a new node into an RBT, the fix-up might include:

1. **Recoloring** (changing a node from red to black or vice versa).
2. **Rotations** (local tree restructuring).

A rotation can change the parent-child relationships, and hence **every node** in the rotated subtree may need a new depth value. The problem is that a rotation near the root can affect **the depths of a large fraction of the entire tree**.

Example Scenario

1. **A Rotation at (or near) the Root**
   - Suppose you have an RBT with n nodes. The root is r.
   - You insert a node in such a way that a **violation** of the red-black properties is detected at the root's level, causing a rotation (or a series of rotations) at or near the root.
2. **Depth Field Update**
   - If the root changes from r to another node r', *all* nodes in the tree effectively have their parent-child path from the new root changed by at least one link.
   - As a result, every node's depth could shift by ± 1 (or some other increment) depending on how the rotation reattaches subtrees.
3. **Potential $O(n)$ Updates**
   - Because the rotation is near the top, a large subtree (potentially most of the nodes in the tree) might now have the wrong depth value and need recalculating.
   - Even though RBT insertion usually only does $O(\log n)$ "fix-up" steps, **each** of those steps could trigger $\Omega(n)$ updates if it occurs near the root.

Hence, **in the worst case**, a single insertion might cause $\Omega(n)$ depth-field updates. Repeated rotations or multiple fix-up steps can multiply this effect, pushing the insertion complexity up to $\Omega(n \log n)$ or at least $\Omega(n)$ per insertion in the worst scenario—certainly higher than the usual $O(\log n)$ bound without this augmentation.

3.4 Conclusion

Storing and maintaining a node's *depth* in a red-black tree can cause large-scale updates after certain rotations (especially those near the root). Because these updates can cascade through a significant portion of the tree, **the worst-case insertion time increases** beyond the usual $O(\log n)$.

In other words, **augmenting each node with a "depth" field** that must be kept accurate in all operations forces potentially $\Omega(n)$ updates in a single insertion, thereby **increasing the asymptotic insertion complexity**.