Underscore

When Name Mangling Happens

1. **Double Leading Underscore + No Trailing Underscore**
   ✅ **Mangled**: __variable ➡️ _ClassName__variable
2. **Double Leading Underscore + One Trailing Underscore**
   ✅ **Mangled**: __variable_ ➡️ _ClassName__variable_
3. **Double Leading Underscore + Two or More Trailing Underscores**
   ❌ **NOT Mangled**: __variable__ ➡️ No name change (these are special/magic methods like __init__, __str__, etc.)
4. **Single Underscore** (e.g., _variable)
   ❌ **NOT Mangled**: Single underscores are just a convention to indicate "internal use only", but Python does not mangle them.

Big-O Notation

- **Big-O**: Your speed is **never faster** than t(n) — you stay within the speed limit.
- **Omega**: Your speed is **never slower** than t(n) — you drive at least this fast.
- **Theta**: Your speed is **exactly matched** with t(n) — you drive at a constant speed that stays

Sorting

Comparison-Based Sorting Algorithms

Insertion Sort

- **Stability:** It is indeed stable by default. Each insertion step places an element into its correct position without changing the relative order of identical elements.
- **Complexity:**
  - **Best Case:** O(n) when the array is already sorted.
  - **Average Case:** $O(n^2)$ due to the nested shifting operations.
  - **Worst Case:** $O(n^2)$, typically when the array is in reverse order.

Merge Sort

- **Stability:** Merge Sort is often considered stable if it's implemented using a stable merging procedure. Standard merge operations do not reorder elements unnecessarily, so it's stable in most common implementations.
- **Complexity:** O(n log n) in the best, average, and worst cases.

Quick Sort

- **Stability:** The conventional in-place Quick Sort is not stable because swapping elements around a pivot can change the relative order of identical elements.
- **Complexity:**
    - **Best/Average:** O(n log n)
    - **Worst:** O(n^2) when chosen pivots are consistently poor (e.g., always picking the largest or smallest element as pivot).
- A stable variant of Quick Sort can be implemented, but it usually involves additional data structures. Its asymptotic complexities remain the same, though in practice it may be less efficient due to the extra overhead.

Randomized Quick Sort

- **Stability:** Not stable by default for the same reasons as normal Quick Sort.
- **Complexity:** Randomization provides a high probability of O(n log n) for both best and average cases and O(n^2) in the worst case. Stability would again require additional effort.

Heap Sort

- **Stability:** Traditional Heap Sort is not stable because the heap construction and extraction process can disrupt the relative order of equal elements.
- **Complexity:** O(n log n) for best, average, and worst cases.
- Stable variants of Heap Sort exist but are not as straightforward and tend to be more complex in practice. They retain O(n log n) complexity.
- **Ascending vs. Descending Order:**
    - To sort in descending order, use a max-heap.
    - To sort in ascending order, use a min-heap.

Non-Comparison-Based Sorting Algorithms

Counting Sort

- **Stability**: Counting Sort is inherently stable if implemented using a cumulative count array and placing elements into the output array in a stable manner.
- **Complexity:** O(n + k), where k is the range of the input values.

Radix Sort

- **Stability:** Radix Sort relies on a stable subroutine (often Counting Sort) for sorting each digit or character. If the sub-sort is stable, then Radix Sort is stable.
- **Complexity:** O(d(n + k)), where d is the number of digits (or passes) and k is the range of each digit.

Bucket Sort

- **Stability:** Bucket Sort can be stable if the sorting algorithm used within each bucket is stable.
- **Complexity:** Usually O(n + k) in the best and average cases (assuming a uniform distribution of elements into buckets), but can degrade to O(n^2) in the worst case if the distribution is not uniform or if one bucket ends up containing most of the elements.

Strassen

Empirical studies have shown that Strassen's algorithm only outperforms naive matrix multiplication for **very large matrix sizes** (typically n > 1024 or n > 2048, depending on hardware).

- Strassen's method becomes more efficient due to its asymptotic complexity $O(n^{log_2 7})$.

Order Statistics

Sort, then pick the k-th element. Linear time also works if only one element is needed.

Trees

1. Binary Search Tree (BST)

- **Structure:** A BST arranges elements so that for any node, all keys in the left subtree are smaller and all keys in the right subtree are larger.
- **Height:**
    - Ideal (balanced) BST: Height is about O(log n).

    - Non-ideal (skewed) BST: Height can degrade to O(n), basically becoming like a linked list if you keep inserting elements in sorted order.
- **Operations:**
    - **Insertion:**
        - Best/Average (balanced BST): O(log n)
        - Worst (skewed tree): O(n)
    - **Deletion:**
        - Best/Average (balanced): O(log n)
        - Worst (skewed): O(n)
    - **Search:**
        - Best/Average (balanced): O(log n)
        - Worst (skewed): O(n)
- **Merging Two BSTs:**
    - One approach: Do an in-order traversal of both (O(n + m)), merge the resulting sorted lists (O(n + m)), and then build a balanced BST from that merged list (O(n + m)). Overall: O(n + m).
    - If you just insert one tree's elements into another naïvely, complexity can be O(m log(n+m)) if the trees are balanced, but potentially O(mn) in the worst case if skewed.

---

2. AVL Tree (Self-Balanced BST)

- **Properties:**
    - For any node, the height difference between its left and right subtree is at most 1.
    - Rotations are used after insertions or deletions to maintain balance.
    - Height remains O(log n).
- **Operations:**
    - **Insertion:** O(log n)
    - **Deletion:** O(log n)
    - **Search:** O(log n)

- **Merging Two AVL Trees:**
    - Like a BST, if you have their sorted traversals, you can merge those lists and then construct a balanced AVL tree in O(n + m).
    - Inserting all elements from one tree into another using normal insert: O(m log(n+m)).

---

3. Red-Black Tree (Self-Balanced BST)

- **Properties & Rules:**
    1. Every node is either red or black.
    2. The root is always black.
    3. No two adjacent red nodes (a red node cannot have a red parent).
    4. Every path from a node to its descendant NIL leaves has the same number of black nodes.

These rules ensure the tree remains roughly balanced, with height O(log n).

- **Operations:**
    - **Insertion:** O(log n). You insert like a BST, then fix violations using rotations and recoloring.
    - **Deletion:** O(log n). Similar to insertion, you remove the node and fix color/rotation issues.
    - **Search:** O(log n)
- **Merging Two Red-Black Trees:**
    - Typically done by inserting all elements of one into the other: O(m log(n+m)).
    - There are specialized merge operations but they're not as commonly discussed at an undergrad level.

In a Red-Black Tree (RB Tree), the height is kept "balanced" through the red-black properties. One often-stated characteristic is the relationship between the shortest and longest paths from the root to a leaf:

- **Shortest Path:**
  Consider any path from the root to a leaf that has the minimum number of nodes. In a Red-Black Tree, every root-to-leaf path contains the same number of black nodes (this is part of the Red-Black tree properties). The shortest path you can have is essentially one

that goes through the fewest nodes while still including all those black nodes. Since red nodes are optional (you don't need them to maintain black-height), the shortest path can be thought of as a path that is "all black" or nearly so.

- **Longest Path:**
  The longest path from the root to a leaf in a Red-Black Tree can have red nodes interspersed between the black nodes. Because no two red nodes can appear consecutively, the longest path can at most "double" the number of nodes compared to the shortest path by alternating black and red nodes.

---

4. Heap (Binary Heap)

- **Properties:**
  - A complete binary tree that satisfies the heap property:
    - Max-heap: each parent ≥ its children.
    - Min-heap: each parent ≤ its children.
  - Usually represented as an array.
- **Operations:**
  - **Insertion:** O(log n), due to "bubble-up" to maintain heap order.
  - **Deletion (Extract-Min/Max):** O(log n), due to "heapify-down" after removing the root.
  - **Search:** O(n), since you'd need to scan all elements (no BST-like ordering).
- **Merging Two Heaps:**
  - If you combine two arrays and then call build-heap on the combined array, it's O(n + m).
  - If you insert elements one by one, that's O(m log(n+m)).

---

5. Hash Table

- **Structure:**
  - Uses a hash function to compute an index in an array where the element should go.
  - Collisions occur when two elements map to the same index.

- **Operations (average with a good hash and low load factor):**
  - **Insertion:** O(1) average, O(n) worst if collisions become excessive or rehashing is triggered.
  - **Deletion:** O(1) average, O(n) worst for the same reasons.
  - **Search:** O(1) average, O(n) worst.
- **Merging Two Hash Tables:**
  - Usually done by re-inserting elements from one into the other. On average O(n) if hash inserts are O(1).
  - Worst case can degrade if collisions are extreme.

Open Addressing in Hash Tables:

- Instead of separate chaining (linking collisions into lists), all elements are stored in the main table itself.
- **Techniques:**
  - **Linear Probing:** If a spot is taken, try the next one, and so forth.
    - Simple but prone to "primary clustering," where long runs of filled slots form.
  - **Quadratic Probing:** Use increasing quadratic steps (i.e., i^2) to find an empty slot.
    - Reduces clustering somewhat but not fully.
  - **Double Hashing:** Use a second hash function to determine the step size.
    - Minimizes clustering better than linear or quadratic probing.
- **Properties for Open Addressing:**
  - Performance heavily depends on load factor (the ratio of elements to table size).
  - High load factor → long probe sequences → performance degrades.
  - Periodic rehashing to a larger table helps keep the average operations near O(1).

Hash Function After Resizing:

- The core hash function remains the same.
- Only the modulus (table size) changes, altering how the hash values map to table indices.

Chaining vs Open Addressing

Open addressing saves space than chaining, for it doesn't need the extra space to store linked list pointers. On the other hand, having empty buckets is also wasting space. Whether a hash table with open addressing is better than chaining depends on the size of items and cache effectiveness.

In general, if an item takes much more spance than a pointer, then open addressing with dynamic tabling would take up much more space during table expansion than chaning, for each new bucket must have the size to store an item. If the size of items is small (e.g. the keys and the values are just pointers), then open addressing would be better.

Up until now, we often treat RAM as the main memory for storing a data structure, and have ignored the use of cache. However, there is a major difference in accessing speed between these two types of hardware: Accessing RAM takes about 100 ns while accessing cache takes only about 1 ns. Thus, in practice, we would want a data structure to be more cache effective. As a general rule, a data structure is more cache-effective if the items often used together are placed close to each other in RAM. Among all the probing schemes for open addressing, linear probing follows this rule better. In general, open addressing is more cache effective than chaining, for items in chaining are in separately allocated space.

Hash tables vs Balanced Binary Search Trees

The choice between hash tables and balanced binary search trees (BSTs) depends on the specific requirements and characteristics of the practical application. Each data structure has its own strengths and weaknesses. Both data structures support the following:

1. Good for Key-Value Pairs: Both are good for key-value pairs where you need to quickly access a value associated with a given key.
2. Dynamic Sizing: Both can dynamically resize to accommodate more elements efficiently.

Here's a comparison of other features to help you make an informed choice:

Hash Tables:

1. Fast Average-Case Lookup: Hash tables provide very fast average-case lookup (search) operations with an average time complexity of O(1) if the hash function is well-designed, and the load factor is kept low.
2. Simple Implementation: Hash tables are relatively easy to implement and use. They are widely available in most programming languages.
3. Not for ordered data or range queries.

4. Not for data traversal in a certain order.

Balanced Binary Search Trees:

1. Ordered Data: BSTs maintain ordered data, making them suitable when data needs to be processed in sorted order.
2. Range Queries: BSTs support efficient range queries, where you can find all elements within a specified range.
3. Flexible Data Sorting: BSTs allow you to easily traverse data in sorted order and can be adapted for various applications, including search, insertion, and deletion.
4. Worst-Case Lookup: BSTs provide a worst-case lookup time complexity of O(log n) in balanced trees, making them efficient for large datasets and worst-case scenarios.

Considerations for Practical Applications:

1. If you need fast and efficient lookup operations with a well-designed hash function and a low load factor, hash tables are often preferred.
2. If you require ordered data or support for range queries, or if you're dealing with worst-case scenarios, balanced BSTs may be a better choice.

AVL vs. RB Trees

Impact of Balancing:

- **AVL Tree:** Height is kept more tightly balanced, typically resulting in a smaller height than a Red-Black Tree for the same number of nodes.
- **Red-Black Tree:** Allows some imbalance, but still guarantees O(log n) height. The tree might be taller on average compared to an AVL.

Operations and Performance:

- **Search:**
    - Both AVL and Red-Black Trees provide O(log n) search time.

- AVL Trees, being more strictly balanced, may offer slightly faster searches in practice due to their generally shorter height.

- **Insertion:**
  - Both are O(log n).
  - **AVL Tree:** After insertion, rotations and height updates can be more frequent because the tree must remain very close to perfectly balanced.
  - **Red-Black Tree:** Insertion may involve fewer rotations on average since the balance conditions are more relaxed.
- **Deletion:**
  - Both achieve O(log n).
  - **AVL Tree:** Similar to insertion, maintaining strict balance may lead to more rotations and updates after deletion.
  - **Red-Black Tree:** Deletion and the subsequent fixing often require carefully handled recoloring and rotations, but it tends to involve fewer rigid conditions than AVL balancing.

Complexity of Rotations:

- **AVL Tree:**
  Each insertion or deletion can cause a cascade of adjustments because the height difference condition is easily violated. However, AVL operations typically require O(1) rotations per insertion or deletion (often 1 or 2 rotations in practice, but checking balance factors up the tree can be needed).
- **Red-Black Tree:**
  Insertion and deletion fixes in Red-Black Trees may involve recoloring and up to a few rotations, but typically the number of rotations is small and bounded by a constant.

Memory and Storage:

- **AVL Tree:** Each node typically stores a balance factor (or subtree height), which can be a small integer. This gives more overhead per node.
- **Red-Black Tree:** Each node stores a color bit (red or black), a smaller overhead than storing full subtree heights.

Use Cases and Preferences:

- **AVL Tree:**
  Ideal when searches are the most frequent operation and very fast lookups are desired. The tree is more rigidly balanced, which can mean fewer tree traversals.
- **Red-Black Tree:**
  Often preferred in system libraries (e.g., C++ std::set, std::map) due to simpler insertion/deletion balancing, less strict height conditions, and potentially better performance in scenarios with lots of inserts and deletes. It strikes a balance between balancing complexity and good enough lookup times.

Robin Hood Hashing

Robin Hood hashing is an open-addressing hash table technique that evens out probe lengths. When inserting, if you encounter a slot with a key that has a shorter probe distance than the new key, you swap them. This ensures longer-traveled keys move closer to their ideal positions, "stealing" spots from those that are closer. By redistributing keys in this way, Robin Hood hashing reduces clustering and improves overall lookup performance.

Dynamic Programming vs. Greedy Algorithm

Dynamic Programming:

1. **Approach**:
   DP involves breaking down a complex problem into smaller overlapping subproblems, solving each subproblem just once, and storing their solutions. Then, DP uses these stored solutions to build up answers to the larger problems. Essentially, DP combines the idea of "divide and conquer" with "memoization" or "tabulation" to ensure that every subproblem's solution is reused and computed efficiently.
2. **Key Characteristics**:
   - **Overlapping Subproblems**: Many subproblems recur multiple times within the solution space.
   - **Optimal Substructure**: The optimal solution to the problem can be constructed from optimal solutions of its subproblems.
   - **State and Recurrence Relations**: DP problems are generally defined by a set of states and transitions (recurrences) that describe how to move from one state to another to ultimately reach the desired solution.
3. **Typical Problems for DP**:
   - **Knapsack Problems**: For example, 0/1 Knapsack, where you must choose items to maximize value without exceeding a capacity.

- **Shortest Path in Graphs**: Problems like the shortest path in Directed Acyclic Graphs can be approached using DP.
- **Sequence Alignment and Editing**: For example, Edit Distance (Levenshtein distance) to transform one string into another.
- **Counting/Combination Problems**: Counting the number of ways to climb stairs, partition numbers, or ways to form certain substrings.
- **Dynamic Scheduling or Partitioning**: Problems like weighted interval scheduling or partitioning sets into subsets with certain properties.

4. **Time Complexity Implication**: Since DP stores solutions to subproblems, it can often reduce exponential brute-force complexities to polynomial time, at the expense of additional memory.

Greedy Algorithms:

1. **Approach**:
   A greedy algorithm makes the "locally optimal" choice at each step, hoping that this will lead to a globally optimal solution. Greedy algorithms do not typically revisit earlier decisions; once a choice is made, it is usually never changed.
2. **Key Characteristics**:
   - **Local Choices**: The algorithm always picks the best local option at the current step.
   - **No Backtracking**: Once a decision is made, the algorithm moves forward; it doesn't systematically consider previously excluded possibilities.
   - **Simple and Fast**: Often easier to implement and runs faster than DP since there's no re-computation or state-saving overhead.
3. **Typical Problems for Greedy Approaches**:
   - **Activity Selection**: Choosing non-overlapping intervals from a set of intervals to maximize the number of tasks or intervals chosen.
   - **Fractional Knapsack**: Unlike the 0/1 Knapsack, the fractional variant can be solved optimally by always taking the item with the highest value-to-weight ratio first.
   - **Minimum Spanning Tree (MST)**: Algorithms like Kruskal's or Prim's that build the MST by always picking the next shortest edge.
   - **Huffman Coding**: Building an optimal prefix-free code by always combining the two smallest frequency trees first.

    - **Shortest Path in Graphs with Nonnegative Edges**: Dijkstra's algorithm can be considered a greedy approach for single-source shortest paths.

4. **Time Complexity Implication**: Greedy algorithms tend to have good performance and low overhead. However, correctness heavily depends on the problem's structure. Not all optimization problems can be optimally solved using a greedy approach.

In Summary:

- **Dynamic Programming** is suitable when the problem can be naturally divided into overlapping subproblems and when solving those subproblems optimally leads to the overall optimal solution.
- **Greedy Algorithms** are suitable when local optimal choices lead to a global optimum and there's a proven "greedy-choice property" for the problem.

Solving LCS with Dynamic Programming

How They Are Related:

1. **Definition of LCS**:
   The Longest Common Subsequence problem involves finding the longest subsequence present in both of two given sequences. A subsequence is defined as a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements.
2. **Complexity**:
   Using DP to solve LCS requires O(m * n) time and space (though space can be optimized to O(min(m, n))), which is significantly more efficient than the naive approach that tries all subsequences.