## Understanding Recurrence Relations

Recurrence relations express the overall running time of a recursive algorithm as a function of its input size. They typically have the form:
[ T(n) = aT\left(\frac{n}{b}\right) + f(n) ]

- **T(n)**: The total amount of work (time complexity) the algorithm does on input size `n`.
- **a**: The number of recursive calls at each level.
- **b**: The factor by which the input size is divided in each recursive call.
- **f(n)**: The amount of work done outside the recursive calls, including divide and combine steps.

## Recursion Tree Insights

A recursion tree helps visualize the work done by each level of the recursive calls. Each level represents the work done at that recursive step, and the sum of all levels' work gives the total work done by the algorithm.

## Identifying Work Pattern and Time Complexity

1. **Sum of Work per Level**:
   - **Increasing**: If the sum of work increases as you move down the tree, the last level dominates the total work.
   - **Decreasing**: If it decreases as you move down, the first level (root) dominates.
   - **Equal**: If the sum of work is equal at all levels, all levels contribute significantly.

## Cheat Sheet for Common Recurrences

- **Binary Search**: (T(n) = T(n/2) + O(1)) ➔ **Decreasing** ➔ (O(\log n))
- **Binary Tree Traversal**: (T(n) = 2T(n/2) + O(1)) ➔ **Equal** ➔ (O(n))
- **Merge Sort**: (T(n) = 2T(n/2) + O(n)) ➔ **Equal** ➔ (O(n \log n))
- **Quick Sort** (average case): (T(n) = 2T(n/2) + O(n)) ➔ **Equal** ➔ (O(n \log n))
- **Multiplying Large Integers (Karatsuba)**: (T(n) = 3T(n/2) + O(n)) ➔ **Increasing** ➔ (O(n^{log_2(3)}))
  Let's discuss the asymptotic running time for each of the mentioned algorithms, which is crucial for understanding their efficiency and performance characteristics.

## 1. Activity Selection Problem

- **Asymptotic Running Time**: (O(n \log n))
- **Explanation**: The most significant part of the algorithm is sorting the activities by their finish times, which takes (O(n \log n)) time. The subsequent greedy selection process is linear, (O(n)), making the overall time complexity (O(n \log n)).

## 2. Assembly Line Scheduling

- **Asymptotic Running Time**: (O(n))
- **Explanation**: Once the input is set up, the algorithm iterates through each station linearly, performing a constant amount of work for each of the (n) stations on two lines. Thus, the running time is linear with respect to the number of stations.

## 3. Coin Changing (Minimum Number of Coins)

- **Asymptotic Running Time**: (O(m \cdot n))
- **Explanation**: For the dynamic programming solution where (m) is the number of coin denominations and (n) is the amount for which we are making change, the algorithm iterates through all coin denominations for each amount up to (n), leading to a time complexity of (O(m \cdot n)).

## 4. Edit Distance

- **Asymptotic Running Time**: (O(m \cdot n))
- **Explanation**: The dynamic programming solution involves filling out a 2D table where (m) and (n) are the lengths of the two strings being compared. Each cell in the table requires constant time to compute, leading to (O(m \cdot n)) time complexity.

## 5. Longest Common Subsequence (LCS)

- **Asymptotic Running Time**: (O(m \cdot n))
- **Explanation**: Similar to the Edit Distance problem, computing the LCS involves filling a 2D table where (m) and (n) are the lengths of the two sequences. Since each cell computation takes constant time, the overall time complexity is (O(m \cdot n)).

## 6. Fibonacci Sequence

- **Dynamic Programming (Bottom-Up Approach) Running Time**: (O(n))
- **Explanation**: When using dynamic programming (specifically the bottom-up approach) to compute the (n^{th}) Fibonacci number, the algorithm iterates from the base cases up to (n), performing a constant amount of work for each number, resulting in a linear time complexity.

## 7. Chessboard Traversal (Maximal Profit)

- **Asymptotic Running Time**: (O(n^2))
- **Explanation**: For a dynamic programming approach to find the maximal profit path on an (n \times n) chessboard, the algorithm fills a 2D table of size (n \times n), where each cell computation is constant. Thus, the running time is (O(n^2)), proportional to the total number of cells in the table.

These running times provide a high-level view of each algorithm's efficiency, with the understanding that actual performance can also depend on factors such as the implementation details and the specific characteristics of the input data.