Explain Greedy Algorithm with an example

Greedy Algorithm

- Decisions are made based on the information available at the current moment without considering future consequences.
- The key idea is to select the best possible choice at each step.

Example: Coin Change Problem

- Goal: Make a total of $15 using the available coin denominations: $10, $7, $1.
- Optimal Solution: {7, 7, 1} (uses 3 coins).
- Greedy Solution: {10, 1, 1, 1, 1, 1} (uses 6 coins).

This shows that the greedy algorithm does not always yield the optimal solution. In this case, it results in a suboptimal solution (using more coins than necessary).

Explain Asymptotic Notations

O *(Big O)* : Represents Asymptotic Upper Bound i.e., worst case scenario

- {f(n): there exist positive constants c and n0 such that:
  - 0 ≤ f(n) ≤ c·g(n) for all n ≥ n0}

Ω *(Omega)* : Represents Asymptotic Lower Bound i.e., best case scenario

- {f(n): there exist positive constants c and n0 such that:
  - 0 ≤ c·g(n) ≤ f(n) for all n ≥ n0}

θ $(Theta)$: Represents Asymptotic Tight Bound i.e., average case scenario

- {f(n): there exist positive constants c and n0 such that:
  - 0 ≤ c1.g(n) ≤ f(n) ≤ c2.g(n) for all n ≥ n0}

Explain and list Properties of Red-Black Trees

- Red-Black tree is a self-balancing binary search tree in which each node can either be red or black color.
- This data structure requires an extra one- bit color field in each node.

Properties:

- Every node is either red or black.
- The root and leaves (NIL's) are black.
- If a node is red, then its parent is black.
- All simple paths from any node x to a descendant leaf have the same number of black nodes = black-height(x).

Differentiate between B trees and B+ Trees

Aspect / Factor	B-Tree	B+ Tree
Data Storage	All nodes	Leaf nodes only
Key Duplication	Not present	Present (internal nodes)
Leaf Node Linkage	Not linked	Linked sequentially
Sequential Access Speed	Slower	Faster
Tree Height	Shorter (generally)	Taller (generally)
Deletion Operation	More complex	Simpler
Primary Use Case	Random access	Range queries
Key Redundancy	No	Yes
Storage Overhead	Lower	Higher
Modern Database Usage	Less common	Standard

Explain Optimal Storage on Tape with example

- Given n programs P1, P2, …, Pn of length L1, L2, …, Ln respectively.
- It is required to store them on a tape of length "L" such that Mean Retrieval Time (MRT) is a minimum.
- Mean retrieval time of "n" programs is the average time required to retrieve any program.
- In this case, we have to find the permutation of the program order which minimizes the MRT after storing all programs on single tape only.
- Consider three programs (P1, P2, P3) with a length of (L1, L2, L3) = (5, 10, 2).
- Let's find the MRT for different permutations.
- 6 permutations are possible for 3 items.
- The Mean Retrieval Time for each permutation is listed in the following table.

Ordering	Calculation of Total Retrieval Time	Total	(MRT)
P1,P2,P3	5+(5+10)+(5+10+2)=5+15+17	37	37/3≈12.333
P1,P3,P2	5+(5+2)+(5+2+10)=5+7+17	29	29/3≈9.667
P2,P1,P3	10+(10+5)+(10+5+2)=10+15+17	42	42/3=14.000
P2,P3,P1	10+(10+2)+(10+2+5)=10+12+17	39	39/3=13.000
P3,P1,P2	2+(2+5)+(2+5+10)=2+7+17	26	26/3≈8.667
P3,P2,P1	2+(2+10)+(2+10+5)=2+12+17	31	31/3≈10.333

- Greedy algorithm stores the programs on tape in non-decreasing order of their length, which ensures the minimum MRT.

Explain Topological Sorting with an Example

- A topological sort of a DAG = (V, E) is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering.
- Topological Sorting is possible if and only if the graph is a Directed Acyclic Graph.
- There may exist multiple different topological orderings for a given directed acyclic graph.
- Applications:
    - Scheduling jobs from the given dependencies among jobs
    - Instruction Scheduling
    - Determining the order of compilation tasks to perform in make files
    - Data Serialization

Merge Sort Code:

```
Procedure MergeSort(A, lb, ub)
    If lb < ub Then
        mid = (lb + ub) / 2   // Find the middle point

        MergeSort(A, lb, mid)    // Sort first half
        MergeSort(A, mid+1, ub)  // Sort second half
        Merge(A, lb, mid, ub)    // Merge the two halves
    End If
End Procedure

Procedure Merge(A, lb, mid, ub)
    i = lb        // Starting index for left subarray
    j = mid + 1   // Starting index for right subarray
    k = lb        // Starting index for temporary array
    Declare B[0..(ub - lb)]   // Temporary array for merging

    // Merge the two subarrays into B in sorted order
    While i <= mid AND j <= ub
        If A[i] <= A[j] Then
            B[k] = A[i]
            i = i + 1
        Else
            B[k] = A[j]
            j = j + 1
        End If
        k = k + 1
    End While

    // Copy any remaining elements from the left subarray
    While i <= mid
        B[k] = A[i]
        i = i + 1
        k = k + 1
    End While

    // Copy any remaining elements from the right subarray
    While j <= ub
        B[k] = A[j]
        j = j + 1
        k = k + 1
    End While

    // Copy the merged elements from B back to A
    For k = lb to ub
```

```
        A[k] = B[k]
    End For
End Procedure
```

- **Best Case:** O (n log n) O (n log n)
    - The array is always split into two equal halves recursively (depth: log n log n), and each merge step takes O(n)O(n) time.
- **Average Case:** O (n log n) O (n log n)
    - Regardless of the initial order of elements, the division and merging process remains the same.
- **Worst Case:** O (n log n) O (n log n)
    - Even for already sorted or reverse-sorted arrays, the algorithm performs the same number of operations.

Quick Sort Code:

```
Procedure QuickSort(A, low, high)
    If low < high Then
        // Partition the array and get the pivot index
        pivot_index = Partition(A, low, high)

        // Recursively sort elements before and after pivot
        QuickSort(A, low, pivot_index - 1)
        QuickSort(A, pivot_index + 1, high)
    End If
End Procedure

Procedure Partition(A, low, high)
    pivot = A[high]   // Choose the last element as pivot
    i = low - 1       // Index of smaller element

    For j = low to high - 1
        If A[j] <= pivot Then
            i = i + 1
            Swap A[i] and A[j]
        End If
    End For

    Swap A[i + 1] and A[high]   // Place pivot in correct position
    Return i + 1   // Return the pivot index
End Procedure

Procedure Swap(a, b)
    temp = a
    a = b
    b = temp
End Procedure
```

- Best Case: O (n log n) O (n log n) (balanced partitions)
- Average Case: O (n log n) O (n log n)
- Worst Case: O(n^2) O(n2) (unbalanced partitions, e.g., already sorted array)