Design and Analysis of Algorithms

This repository contains the code for the assignments of the course Design and Analysis of Algorithms (19Z402) from PSG College of Technology, Coimbatore.

Table of Contents

DIVIDE AND CONQUER :

- Introduction to Algorithm Design techniques
- Divide and Conquer Methodology
- Solving recurrence relations
- Masters Theorem
- Finding Maximum and Minimum Element
- Quick sort
- Merge sort
- Convex Hull

GREEDY METHOD:

- Greedy Strategy
- Knapsack Problem
- Minimum Spanning Tree
- Single Source Shortest Path Method
- Huffman Trees

DYNAMIC PROGRAMMING :

- Principle of Optimality
- Knapsack Problem
- All Pairs Shortest Path
- Optimal Binary Search Tree
- Multistage Graphs

BACKTRACKING:

- State Space Tree
- Knapsack Problem
- The Eight Queens Problem
- Sum of subsets
- Graph Coloring

BRANCH AND BOUND :

- Bounding Functions
- 0/1 Knapsack Problem

- Traveling SalesPerson Problem
- Assignment Problem

Divide and Conquer Algorithm

Divide and Conquer Algorithm breaks a problem into subproblems that are similar to the original problem, recursively solves the subproblems.
Each subproblem should deal with the same issue as the main problem.
Each recursion of the algorithm makes the problem smaller until it reaches a base case. The algorithm is split into three parts :

- Divide: This divides each problem into smaller problems allowing us to make the number of elements to be calcuated on smaller.
- Conquer: This allows us to perform the required operation on the elements and solve the subproblems by addressing the base case.
- Combine: This is where we recombine all the parts of the problem to find our final result.

Merge Sort

Merge Sort is based on the divide and conquer approach. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

Time Analysis:
Merge Sort Function

- Best Case: O(n log n)
- Average Case: O(n log n)
- Worst Case: O(n log n)

Merge Function

- Best Case: O(n)
- Average Case: O(n)
- Worst Case: O(n)

Quick Sort

Quick Sort is based on the divide and conquer approach.
The idea is to pick the first element as the pivot. We have to pick the second element as i and j as the last element.

- Increment i till we find an element greater than the pivot.
- Decrement j till we find an element less than the pivot.
- Swap the elements at i and j.
- Repeat the process until i is greater than j.

- Swap the pivot with the element at j.

Time Analysis:

- Best Case: O(n log n)
- Average Case: O(n log n)
- Worst Case: O(n^2)

Useful Youtube Vieo:
Quick Sort Algorithm by Abdul Badri

Convex Hull

Convex Hull is the smallest convex polygon that contains all the points in the set.
The algorithm is based on the divide and conquer approach.

- Find the bottem most point.
- Sort the points based on the angle with the bottom most point.
- If two points have the same angle, then the point that is closer to the bottom most point is considered first.
- We insert the first two points into the stack.
- For the remaniang elements, if the angle is counter-clockwise we insert the point into the stack.
- If the angle is clockwise, we pop the top element from the stack and insert the point into the stack.
- The stack contains the convex hull of the points.

Useful Youtube Video:
Convex Hull Algorithm by MIT OCW

Greedy Algorithm

Greedy Algorithm is a technique where we always pick the ***local optima*** in all the steps to find the global optima.
If a problem can be solved by greedy algorithm then we can find that the problem has the following properties:

1. Optimal Substructure: The problem has the property that the optimal solution can be constructed from the optimal solutions of its subproblems.
2. Greedy Choice Property: The problem has the property that a global optimal solution can be arrived at by selecting a local optimal solution.

Minimum Spanning Tree

Minimum Spanning Tree is a tree that connects all the vertices together without any cycles and with the minimum possible total edge weight.
The algorithm is based on the greedy approach. We use the technique to find the paths in the tree to

find the spanning tree (contaning all the vertices) with the minimum weight.
The problem is as follows:

1. You are given an undirected graph with vertices and edges. Each edge has a weight assigned to it.
2. You have to find a graph within the graph that connects all the vertices using the least possible weight.

The objective is to find the algorithm to find MST with complexity nearest to linear time.

Optimal Substructure for MST: **LEMMA 1**

1. if e={u,v} is an edge of some MST.
    - We can contract the edge e by merging u and v into a single vertex.
    - All common connected edges to u and v are connected to the new vertex. They will now become a single edge to the connnected vertex. Out of the edges to the two vertices, we will select the edge with the minimum weight.
    - if T' is the MST of graph G obtained by contracting e, then
      `The graph T' U {e} is an MST of G.`

The possible approaches are:

1. **Brute Force Computation**: This is not feasible as the number of possible spanning trees is very large. This will have a time complexity of O(n!).
2. **Dynamic Programming Approach**:
   In a graph 'G' with 'n' vertices, there are 2^n-2 spanning trees
   The steps are as follows:
    - Guess edge 'e' in a MST
    - Contract edge 'e'
    - Recurse on the contracted graph
    - Uncontract edge 'e'
    - Add 'e' to the MST
    - Return the MST with the minimum weight.
3. **Prim's Algorithm**: In prim's algorithm we start out with a simple cut which is a single vertex. We can take the minimum edge from that cut and add it to the spanning tree and repeat it. We can also pick the edge with the min weight to start (Minimum Cause Vertex).
4. **Kruskal's Algorithm**: In kruskal's algorithm, we always pick the two edges with the least weight. We then start picking edges with the weight increasing slowly and we check each see if they make a cycle. If they do then we don't pick the edge.

Useful Youtube Video:
Minimum Spanning Tree by MIT OCW