9. Recursion

Recursion is when a function calls itself to solve smaller subproblems.

```
# Factorial using recursion - O(n)
def factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * factorial(n-1)

print(factorial(5))  # 120
```

10. Dynamic Programming (Fibonacci)

Dynamic Programming optimizes recursion by storing results of subproblems.

```
# Fibonacci using DP - O(n)
def fib_dp(n):
    dp = [0, 1]
    for i in range(2, n+1):
        dp.append(dp[i-1] + dp[i-2])
    return dp[n]

print(fib_dp(10))  # 55
```

11. Heap (Priority Queue)

Heap is a complete binary tree often implemented as a priority queue.

```
import heapq

# Min Heap - O(log n) for insertion/removal
heap = []
heapq.heappush(heap, 10)
heapq.heappush(heap, 5)
heapq.heappush(heap, 20)

print(heapq.heappop(heap))  # 5
```

12. Trie (Prefix Tree)

Trie is used for efficient prefix-based searching.

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end = True
```

```
    def search(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end

trie = Trie()
trie.insert("cat")
print(trie.search("cat"))  # True
print(trie.search("car"))  # False
```

13. Depth First Search (DFS)

DFS explores as far as possible along each branch before backtracking.

```
graph = {
    'A': ['B', 'C'],
    'B': ['D'],
    'C': ['E'],
    'D': [],
    'E': []
}

# DFS - O(V+E)
def dfs(node, visited=set()):
    if node not in visited:
        print(node, end=" ")
        visited.add(node)
        for neighbor in graph[node]:
            dfs(neighbor, visited)

dfs('A')
```

14. Dijkstra's Algorithm (Shortest Path)

Dijkstra finds the shortest path from a source node to all other nodes in weighted graph.

```
import heapq

def dijkstra(graph, start):
    distances = {node: float('inf') for node in graph}
    distances[start] = 0
    pq = [(0, start)]

    while pq:
        curr_dist, node = heapq.heappop(pq)
        if curr_dist > distances[node]:
            continue
        for neighbor, weight in graph[node]:
            distance = curr_dist + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(pq, (distance, neighbor))
    return distances

graph = {
```

```python
    'A': [('B', 1), ('C', 4)],
    'B': [('C', 2), ('D', 5)],
    'C': [('D', 1)],
    'D': []
}

print(dijkstra(graph, 'A'))  # {'A': 0, 'B': 1, 'C': 3, 'D': 4}
```