1. Aggregate:

```
stack = []
arr = [8,3,1,9,8,7,3,6]
operations = 0
total_cost = 0
unit = []

def push(element):
    global stack,operations,total_cost,unit
    stack.append(element)
    operations+=1
    total_cost+=1
    unit.append(1)
    print("Stack after pushing ",element,": ",stack)

def pop():
    global stack,operations,total_cost,unit
    if len(stack)==0: return
    stack.pop()
    operations+=1
    total_cost+=1
    unit.append(1)

def multipop(n, length):
    for i in range(n):
        pop()
    print("Stack after poping ",n,"elements: ",stack,"\t\t(As",n,"is less than stack length",length,")")

for i in arr:
    print("For element: ",i)
    if i<=len(stack):
        multipop(i, len(stack))
    push(i)

print("T(n) = ",sum(unit)," and Total cost: ",total_cost)
print("Amortized Cost: ",sum(unit)//operations)
```

2. Accounting

```
def accounting(n):
    size = 1
    total = 0
    doubling_cost = 0
    insertion_cost = 0
    balance = 0

    print("Doubling Cost\tInsertion Cost\tTotal Cost\tBalance\t\tSize\n")

    for i in range(1, n+1):
        insertion_cost=1
        if i>size:
            size*=2
            doubling_cost = i-1
        total = doubling_cost+insertion_cost
        balance+=(3-total)
        print(doubling_cost,"\t\t",insertion_cost,"\t\t",total,"\t\t",balance,"\t\t",size)

        doubling_cost = 0
        insertion_cost = 0

accounting(10)
```

3. Potential

```python
doubling_costs = []
current_length = 1
potential = []

for i in range(1, 11):
    if current_length < i:
        current_length *= 2
        doubling_costs.append(i-1)
    else:
        doubling_costs.append(0)
    potential.append(2*i - current_length)

total_cost = [x+1 for x in doubling_costs]

print('Doubling Cost\t Iteration\t Total Cost\t Potential\t Amortized Cost')

print(f'{doubling_costs[0]}\t\t {1}\t\t {total_cost[0]}\t\t {potential[0]}\t\t {total_cost[0] + potential[0]}')

for j in range(1, 10):
    amortized_cost = total_cost[j] + potential[j] - potential[j-1]
    print(f'{doubling_costs[j]}\t\t {j+1}\t\t {total_cost[j]}\t\t {potential[j]}\t\t {amortized_cost}')
```

4. Hiring

```
candidates = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print("Candidates: ", candidates)
interviewed_candidates = []
hired_candidates = []
for candidate in candidates:
    interviewed_candidates.append(candidate)
    if not hired_candidates or candidate > max(hired_candidates):
        hired_candidates.append(candidate)
firing_cost = len(hired_candidates) - 1  # Since the last candidate is the best

print("Normal way : ")
print("Interviewed candidates:", interviewed_candidates)
print("Hired candidates:", hired_candidates)
print("Number of candidates hired:", len(hired_candidates))
print("Firing cost:", firing_cost)

import random

candidates = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
interviewed_candidates = []
hired_candidates = []

print("\nRandomized Approach")
# Randomly select and interview candidates
for i in range(len(candidates)):
    temp = random.choice(candidates)
    interviewed_candidates.append(temp)
    candidates.remove(temp)
max = -1
for i in interviewed_candidates:
    if i > max:
        max = i
        hired_candidates.append(i)
# Calculate firing cost
firing_cost = len(hired_candidates) - 1  # Since the last candidate is the best
print("Interviewed candidates in randomized order:", interviewed_candidates)
print("Hired candidates:", hired_candidates)
print("Number of candidates hired:", len(hired_candidates))
print("Firing cost:", firing_cost)
```

5. Lumoto (QS)

```python
import random
import time

def lumoto_partitoning(low, high, data):
    i = low
    j = low
    pivot = data[high]

    while j<high:
        if data[j]<pivot:
            data[i], data[j] = data[j],data[i]
            i+=1
        j+=1

    data[i],data[high] = data[high], data[i]
    return i

def quick_sort(low, high, data):
    if low<high:
        p = lumoto_partitoning(low, high, data)
        quick_sort(low, p-1, data)
        quick_sort(p+1, high, data)

def randomized_partitioning(low, high, data):
    rand_index = random.randint(low, high)
    data[rand_index], data[high] = data[high], data[rand_index]
    return lumoto_partitoning(low, high, data)

def quick_sort_rand(low, high, data):
    if low<high:
        p = randomized_partitioning(low, high, data)
        quick_sort(low, p-1, data)
        quick_sort(p+1, high, data)

data = [i for i in range(900,0,-1)]
start = time.time()
quick_sort(0,len(data)-1,data)
end = time.time()
print("Time taken (Lumoto): ",end-start)
```

```
data = [i for i in range(900,0,-1)]
start = time.time()
quick_sort_rand(0,len(data)-1,data)
end = time.time()
print("Time taken (Lumoto with Randomization): ",end-start)
```

6. Hoarse (QS):

```
import random
import time

def hoare_partitioning(low, high, data):
    i = low-1
    j = high+1
    pivot = data[low]
    while True:

        i+=1
        if data[i]<pivot:
            i+=1

        j-=1
        if data[j] >pivot:
            j-=1

        if i>=j:
            return j

        data[i], data[j] = data[j], data[i]

def quick_sort(low, high, data):
    if low<high:
        p = hoare_partitioning(low,high,data)
        quick_sort(low, p, data)
        quick_sort(p+1, high, data)

def randomized_partitioning(low, high, data):
    random_index = random.randint(low, high)
    data[random_index], data[low] = data[low], data[random_index]
    return hoare_partitioning(low, high, data)
```

```
def quick_sort_rand(low, high, data):
    if low<high:
        p = randomized_partitioning(low,high,data)
        quick_sort(low, p, data)
        quick_sort(p+1, high, data)

data = [i for i in range(900, 0, -1)]
start = time.time()
quick_sort(0, len(data)-1, data)
end = time.time()
print("Time Taken (Hoare): ",end-start)

data = [i for i in range(900, 0, -1)]
start = time.time()
quick_sort_rand(0, len(data)-1, data)
end = time.time()
print("Time Taken (Hoare with Randomization): ",end-start)
```

7. KD-Tree (Balanced):

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.depth = 0
        self.left = None
        self.right = None

def traverse_in_order(curr):
    if curr is None:
        return
    traverse_in_order(curr.left)
    print(f"({', '.join(map(str, curr.data))}) ", end="")
    traverse_in_order(curr.right)

def make_kd_tree(seq, depth=0):
    if len(seq) == 0:
        return None

    k = len(seq[0]) #no of dimensions
    dim = depth % k

    seq.sort(key=lambda x: x[dim])
    mid = len(seq) // 2
    mid_elem = seq[mid]

    root = Node(mid_elem)
    left_sub_arr = seq[:mid]
    right_sub_arr = seq[mid+1:]

    root.depth = depth
    root.left = make_kd_tree(left_sub_arr, depth+1)
    root.right = make_kd_tree(right_sub_arr, depth+1)

    return root

seq = [[6,2], [7,1], [2,9], [3,6], [4,8], [8,4], [5,3], [1,5], [9,5]]
root = make_kd_tree(seq)
print("Inorder Traversal: ",end='')
traverse_in_order(root)
```

8. KD-Tree (Unbalanced)

```
class KDNode:
    def __init__(self, data):
        self.data= data
        self.depth = 0
        self.left = None
        self.right = None

def insert(node,point):
    if node is None:
        return KDNode(point)
    dim=node.depth%2

    if point[dim] < node.data[dim]:
        node.left=insert(node.left,point)
        node.left.depth=node.depth+1
    else:
        node.right=insert(node.right,point)
        node.right.depth=node.depth+1

    return node

def inorder(node):
    if node is None:
      return
    inorder(node.left)
    print(f"({','.join(map(str,node.data))}) ",end="")
    inorder(node.right)

unbalanced_points = [[6, 2], [7, 1], [2, 9], [3, 6], [4, 8], [8, 4], [5, 3], [1, 5], [9, 5]]
unbalanced_root = KDNode(unbalanced_points[0])

for point in unbalanced_points[1:]:
  insert(unbalanced_root, point)

print("Initial tree:")
inorder(unbalanced_root)
insert(unbalanced_root, [3,5])
print("\nAfter insertion of point (3,5):")
inorder(unbalanced_root)
```

9. Ford-Fulkerson:

```
class Graph:
    def __init__(self, graph):
        self.graph = graph
        self.ROW = len(graph)



    def bfs(self, s, t, parent):
        visited = [False] * self.ROW
        queue = []
        queue.append(s)
        visited[s] = True

        while queue:

            u = queue.pop(0)

            for ind, val in enumerate(self.graph[u]):
                if not visited[ind] and val > 0:
                    queue.append(ind)
                    visited[ind] = True
                    parent[ind] = u
        return visited[t], parent

    def ford_fulkerson(self, source, sink):
        max_flow = 0
        parent = [-1] * self.ROW

        while True:
            found_path, parent = self.bfs(source, sink, parent)
            if not found_path:
                break

            path_flow = float("Inf")
            s = sink
            while s != source:
                path_flow = min(path_flow, self.graph[parent[s]][s])
                s = parent[s]
            max_flow += path_flow
```

```
            # Print the augmented path and its minimum value
            path = [sink]
            v = sink
            while v != source:
                u = parent[v]
                path.insert(0, u)
                v = u
            print("Augmented path: ", " -> ".join(str(x) for x in path), " Minimum flow: ", path_flow)
            v = sink
            while v != source:
                u = parent[v]
                self.graph[u][v] -= path_flow
                self.graph[v][u] += path_flow
                v = u

        return max_flow

#             0   1   2   3   4   5
#             S,  B,  P,  M,  K,  D
graph = [   [0,  0,  0,  0,  0,  0],  #S
            [17, 0,  0,  0,  10, 0],  #B
            [6,  7,  0,  0,  0,  0],  #P
            [0,  12, 0,  0,  14, 0],  #M
            [0,  0, 10,  6,  0,  0],  #K
            [0,  0,  0,  8,  14, 0],  #D
        ]

g = Graph(graph)
source = 5
sink = 0
print("Max Flow: %d " % g.ford_fulkerson(source, sink))
```

10. Convex Hull

```python
import math
def structure(p, q, r):
    v = (q[1] - p[1]) * (r[0] - q[0]) - (q[0] - p[0]) * (r[1] - q[1])

    if v == 0:
        return 0 # 0 hai to collinear
    return 1 if v > 0 else -1 # 1=clockwise, -1=counterclockwise

def gs(points):
    n = len(points)
    if n < 3:
        return []

    min_pt = min(points, key=lambda x: (x[1], x[0]))

    angle_key = lambda x: math.atan2(x[1] - min_pt[1], x[0] - min_pt[0])
    sort_pts = sorted(points, key=lambda x: (angle_key(x), x))

    print(sort_pts)

    stack = [sort_pts[0], sort_pts[1], sort_pts[2]]
    print(f"After addition 3 points , stack : {stack}")

    for i in range(3, n):

        while len(stack) > 1 and structure(stack[-2], stack[-1], sort_pts[i]) == 1:
            stack.pop()

        stack.append(sort_pts[i])
        print(f"adding {sort_pts[i]} --> stack : {stack}")

    return stack

points =[(0, 0), (3, 0), (1, 4), (3, 3), (5, 2), (5, 5), (9, 6), (7, 0),(10,0)]
print(f"Convex hull will be : {gs(points)}")
```