DESIGN AND ANALYSIS OF ALGORITHMS LAB

TOPIC 5: GREEDY ALGORITHMS (Q1–Q12)

Q1. Activity Selection Problem

Aim: To select maximum non-overlapping activities.

Procedure: Sort by finish time and select greedily.

Code:

```
def activity_selection(start, finish):
    activities = sorted(zip(start, finish), key=lambda x: x[1])
    selected = [activities[0]]
    last = activities[0][1]
    for i in range(1, len(activities)):
        if activities[i][0] >= last:
            selected.append(activities[i])
            last = activities[i][1]
    return selected

print(activity_selection([1,3,0,5,8,5],[2,4,6,7,9,9]))
```

Output: Selected activities displayed.

Result: Maximum activities selected successfully.

Q2. Fractional Knapsack

Aim: To maximize profit allowing fractions.

Procedure: Sort items by value/weight ratio.

Code:

```
def fractional_knapsack(W, items):
    items.sort(key=lambda x: x[1]/x[0], reverse=True)
    profit = 0
    for wt, val in items:
        if W >= wt:
            profit += val
            W -= wt
        else:
            profit += val * (W/wt)
            break
    return profit

print(fractional_knapsack(50, [(10,60),(20,100),(30,120)]))
```

Output: 240.0

Result: Maximum profit obtained.

Q3. Job Sequencing with Deadlines

Aim: To schedule jobs to maximize profit.

Procedure: Sort jobs by profit and schedule greedily.

Code:

```
def job_sequencing(jobs):
    jobs.sort(key=lambda x: x[2], reverse=True)
    max_dead = max(j[1] for j in jobs)
    slot = [-1]*(max_dead+1)
    profit = 0
    for j in jobs:
        for d in range(j[1],0,-1):
            if slot[d] == -1:
                slot[d] = j[0]
                profit += j[2]
                break
    return profit
jobs=[('A',2,100),('B',1,19),('C',2,27),('D',1,25),('E',3,15)]
```

```
print(job_sequencing(jobs))
```

Output: 142

Result: Jobs scheduled successfully.

Q4. Huffman Coding

Aim: To generate optimal prefix codes.

Procedure: Merge lowest frequency symbols greedily.

Code:

```
import heapq
def huffman(freq):
    heap=[[w,[c,""]] for c,w in freq.items()]
    heapq.heapify(heap)
    while len(heap)>1:
        lo=heapq.heappop(heap)
        hi=heapq.heappop(heap)
        for p in lo[1:]:
            p[1]='0'+p[1]
        for p in hi[1:]:
            p[1]='1'+p[1]
        heapq.heappush(heap,[lo[0]+hi[0]]+lo[1:]+hi[1:])
    return heap[0][1:]

print(huffman({'a':5,'b':9,'c':12,'d':13,'e':16,'f':45}))
```

Output: Huffman codes printed.

Result: Optimal codes generated.

Q5. Prim’s Algorithm

Aim: To find Minimum Spanning Tree.

Procedure: Select minimum edge connecting vertices.

Code:

```
import heapq
def prim(graph):
    visited={0}
    edges=[(w,0,v) for v,w in graph[0]]
    heapq.heapify(edges)
    cost=0
    while edges:
        w,u,v=heapq.heappop(edges)
        if v not in visited:
            visited.add(v)
            cost+=w
            for nv,nw in graph[v]:
                if nv not in visited:
                    heapq.heappush(edges,(nw,v,nv))
    return cost
```

Output: MST cost printed.

Result: MST constructed successfully.

Q6. Kruskal’s Algorithm

Aim: To find MST using greedy approach.

Procedure: Sort edges and avoid cycles.

Code:

```
def kruskal(edges,n):
    parent=list(range(n))
    def find(x):
        if parent[x]!=x:
            parent[x]=find(parent[x])
        return parent[x]
    cost=0
    for u,v,w in sorted(edges,key=lambda x:x[2]):
```

```
            if find(u)!=find(v):
                parent[find(u)]=find(v)
                cost+=w
        return cost
```

Output: MST cost printed.

Result: MST obtained successfully.

Q7. Dijkstra's Algorithm

Aim: To find shortest path from source.

Procedure: Select vertex with minimum distance.

Code:

```
import heapq
def dijkstra(graph,src):
    dist={v:float('inf') for v in graph}
    dist[src]=0
    pq=[(0,src)]
    while pq:
        d,u=heapq.heappop(pq)
        for v,w in graph[u]:
            if d+w<dist[v]:
                dist[v]=d+w
                heapq.heappush(pq,(dist[v],v))
    return dist
```

Output: Shortest paths printed.

Result: Shortest path found successfully.

Q8. Optimal Merge Pattern

Aim: To minimize total merging cost.

Procedure: Merge smallest files first.

Code:

```
import heapq
def optimal_merge(files):
    heapq.heapify(files)
    cost=0
    while len(files)>1:
        a=heapq.heappop(files)
        b=heapq.heappop(files)
        cost+=a+b
        heapq.heappush(files,a+b)
    return cost

print(optimal_merge([20,30,10,5]))
```

Output: 115

Result: Optimal merge achieved.

Q9. Coin Change (Greedy)

Aim: To form amount with minimum coins.

Procedure: Pick largest denomination first.

Code:

```
def coin_change(coins,amt):
    res=[]
    for c in sorted(coins,reverse=True):
        while amt>=c:
            amt-=c
            res.append(c)
    return res

print(coin_change([1,2,5,10],27))
```

Output: [10, 10, 5, 2]

Result: Amount formed successfully.

Q10. Minimum Platforms

Aim: To find minimum number of platforms.

Procedure: Sort arrival and departure times.

Code:

```
def min_platform(arr,dep):
    arr.sort(); dep.sort()
    i=j=0; plat=ans=0
    while i<len(arr) and j<len(dep):
        if arr[i]<=dep[j]:
            plat+=1; ans=max(ans,plat); i+=1
        else:
            plat-=1; j+=1
    return ans

print(min_platform([900,940,950,1100],[910,1200,1120,1130]))
```

Output: 2

Result: Minimum platforms calculated.

Q11. Optimal Storage on Tapes

Aim: To minimize mean retrieval time.

Procedure: Sort programs by length.

Code:

```
def optimal_storage(files):
    files.sort()
    total=0; s=0
    for f in files:
        s+=f; total+=s
    return total/len(files)

print(optimal_storage([5,10,3]))
```

Output: Mean retrieval time printed.

Result: Optimal storage achieved.

Q12. Scheduling with Deadlines

Aim: To schedule tasks efficiently.

Procedure: Schedule tasks greedily based on deadlines.

Code:

```
def schedule(tasks):
    tasks.sort(key=lambda x:x[1])
    time=0
    for t,d in tasks:
        if time+t<=d:
            time+=t
    return time

print(schedule([(2,4),(1,1),(3,5)]))
```

Output: Total time printed.

Result: Tasks scheduled successfully.