Third Year B. Tech., Sem V 2022-23

Design and Analysis of Algorithm Lab

Assignment / Journal submission

PRN/ Roll No: 21520010

Full name: Mayur Sunil Savale

Batch: T8

Assignment: Week 6

Title of assignment: Greedy Method

To apply Greedy method to solve problems of

1. Job sequencing with deadlines

   A) Generate table of feasible, processing sequencing, profit.

   Ans:

   a) Algorithm: (Pseudocode)

   - In a loop, we run the array elements and add id, deadline and profit as keys and values to the dictionary.
   - We append these dictionaries to a list
   - We sort the list based on profit earned in descending order.

   b) Code snapshots of implementation

```
n1=7
profits=[3,5,20,18,1,6,30]
deadlines=[1,3,4,3,2,1,2]
jobs=[]

for i in range(n1):
```

```
    tmp={
        "id": i+1,
        "profit": profits[i],
        "deadline": deadlines[i],
        "taken": False
        }
    jobs.append(tmp)

jobs=sorted(jobs,key=lambda k:(-k['profit']))

for job in jobs:
    print("id - "+str(job["id"]))
    print("profit - "+str(job["profit"]))
    print("deadline - "+str(job["deadline"]))
    print()
```

Output:

```
id - 7
profit - 30
deadline - 2

id - 3
profit - 20
deadline - 4

id - 4
profit - 18
deadline - 3

id - 6
profit - 6
deadline - 1

id - 2
profit - 5
deadline - 3

id - 1
profit - 3
deadline - 1

id - 5
profit - 1
deadline - 2
```

In Text :

```
runfile('C:/Users/mayur/OneDrive/Desktop/pyhton
program/untitled1.py',
wdir='C:/Users/mayur/OneDrive/Desktop/pyhton program')
id - 7
profit - 30
deadline - 2

id - 3
profit - 20
deadline - 4

id - 4
profit - 18
deadline - 3

id - 6
profit - 6
deadline - 1

id - 2
profit - 5
deadline - 3

id - 1
profit - 3
deadline - 1

id - 5
profit - 1
deadline - 2
```

c) Complexity of proposed algorithm (Time & Space)

- Time Complexity: O(N log N)
- Space Complexity: O(N)

d) Your comment (How your solution is optimal?)

The solution is straightforward implementation with optimality of the lambda function for custom sort

B) What is the solution generated by the fraction JS when n=7, (p1, p2, …., p7) = (3, 5, 20, 18, 1, 6, 30), and (d1, d2, d3, …., d7) = (1, 3, 4, 3, 2, 1, 2)?

Ans:

a) Algorithm: (Pseudocode)

- Sort all jobs in descending order of profit.
- Iterate on jobs in descending order of profit. For each job, do the following:
  a. Find a time slot i, such that slot is empty and i < deadline and i is greatest. Put the job in this slot and mark this slot filled.
  b. If no such i exists, then ignore the job.
- Output the sequence list with the maximised profit of the list.

b) Code snapshots of implementation

```
def printJobScheduling(jobs,t):

  n=len(jobs)

  #reverse sort profit
  jobs= sorted(jobs,key=lambda k: (-k['profit']))

  result=[False]*t

  res=['1']*t

  maxiProfit=0

  for job in jobs:
    for j in range(min(t-1,job["deadline"]-1),-1,-1):
      if result[j] is False:
```

```
                result[j]=True
                res[j]=job["id"]
                maxiProfit+=job["profit"]
                break
    return res,maxiProfit

n1=7
profits=[3,5,20,18,1,6,30]
deadlines=[1,3,4,3,2,1,2]
jobs=[]
maxiDeadline=max(deadlines)

for i in range(n1):
    tmp={
        "id": i+1,
        "profit": profits[i],
        "deadline": deadlines[i],
        "taken": False
        }
    jobs.append(tmp)

result,maxProfit=printJobScheduling(jobs,maxiDeadline)

print("Job Schedule Sequence: ",end='')
print(result)

print("Maximised Profit: "+str(maxProfit))
```

Output:

```
In [1]: runfile('C:/Users/mayur/OneDrive/Desktop/pyhton program/untitled0.py',
wdir='C:/Users/mayur/OneDrive/Desktop/pyhton program')
Job Schedule Sequence: [6, 7, 4, 3]
Maximised Profit: 74

In [2]: runfile('C:/Users/mayur/OneDrive/Desktop/pyhton program/untitled0.py',
wdir='C:/Users/mayur/OneDrive/Desktop/pyhton program')
Job Schedule Sequence: [6, 7, 4, 3]
Maximised Profit: 74
```

c) **Complexity of proposed algorithm (Time & Space)**

- Time Complexity: $O(N^2)$
- Space Complexity: $O(N)$

d) **Your comment (How your solution is optimal?)**

The solution uses a greedy approach which gets the optional result in $O(N^2)$ complexity. This may not be the most efficient, but the solution is robust.

C) **Input:** Five Jobs with following deadlines and profits

JobID	Deadlines	Profit
a	2	100
b	1	19
c	2	27
d	1	25
e	3	15

Output: Following is maximum profit sequence of jobs:

c, a, e

a) Algorithm: (Pseudocode)

- Sort all jobs in descending order of profit.
- Iterate on jobs in descending order of profit. For each job, do the following:
  a. Find a time slot i, such that slot is empty and i < deadline and i is greatest. Put the job in this slot and mark this slot filled.
  b. If no such i exists, then ignore the job.
- Output the sequence list with the maximised profit of the list.

b) Code snapshots of implementation

```
def printJobScheduling(jobs,t):

    n=len(jobs)

    #reverse sort profit
    jobs= sorted(jobs,key=lambda k: (-k['profit']))

    result=[False]*t

    res=['1']*t

    maxiProfit=0

    for job in jobs:
```

```
        for j in range(min(t-1,job["deadline"]-1),-1,-1):
            if result[j] is False:
                result[j]=True
                res[j]=job["id"]
                maxiProfit+=job["profit"]
                break
    return res,maxiProfit


jobs=[
    {
        "id": "a",
        "deadline": 2,
        "profit": 100
    },
    {
        "id": "b",
        "deadline": 1,
        "profit": 19
    },
    {
        "id": "c",
        "deadline": 2,
        "profit": 27
    },
    {
        "id": "d",
        "deadline": 1,
        "profit": 25
    },
    {
        "id": "e",
        "deadline": 3,
        "profit": 15
    }
```

```
]
deadlines=[]

for job in jobs:
   deadlines.append(job["deadline"])

maxiDeadline=max(deadlines)

result,maxProfit=printJobScheduling(jobs,maxiDeadline)

print("Job Schedule Sequence: ",end=' ')
print(result)

print("Maximised Profit: "+str(maxProfit))
```

Output:

```
In [4]: runfile('C:/Users/mayur/OneDrive/Desktop/pyhton program/untitled2.py', wdir='C:/Users/
mayur/OneDrive/Desktop/pyhton program')
Job Schedule Sequence:  ['c', 'a', 'e']
Maximised Profit: 142

In [5]:
```

c) **Complexity of proposed algorithm (Time & Space)**

- Time Complexity: $O(N^2)$
- Space Complexity: $O(N)$

d) **Your comment (How your solution is optimal?)**

The solution is straightforward implementation with optimality of the lambda function for custom sort. We essentially are working to fill each slot with the best possible subject to the deadline.

D) Study and implement Disjoint set algorithm to reduce time complexity of JS from $O(n^2)$ to nearly $O(n)$

Ans:

a) Algorithm: (Pseudocode)

- Sort all jobs in descending order of profit.
- Iterate on jobs in descending order of profit. For each job, do the following:
    c. Find a time slot i, such that slot is empty and i < deadline and i is greatest. Put the job in this slot and mark this slot filled.
    d. If no such i exists, then ignore the job.
- Output the sequence list with the maximised profit of the list.

b) Code snapshots of implementation

```
import sys

class DisjointSet:
    def __init__(self, n):
        self.parent = [i for i in range(n+1)]

    def find(self, s):
        if s == self.parent[s]:
            return s
        self.parent[s] = self.find(self.parent[s])
        return self.parent[s]

    def merge(self, u, v):
        self.parent[v]=u

def cmp(a):
    return a['profit']

def maxDeadline(jobs, n):
    ans = - sys.maxsize - 1
    for i in range(n):
        ans=max(ans, jobs[i]['deadline'])
    return ans
```

```
def printScheduling(jobs, n):
    jobs = sorted(jobs, key= cmp, reverse = True)

    #create a disjoint set data structure
    max_deadline = maxDeadline(jobs, n)
    ds=DisjointSet(max_deadline)
    maxiProfit=0
    for i in range(n):
            #find maximum available free slot
            available_slot=ds.find(jobs[i]['deadline'])
            if(available_slot > 0):
                    ds.merge(ds.find(available_slot-1),available_slot)
                    print(jobs[i]['id'], end=" ")
                    maxiProfit += jobs[i]['profit']
    return maxiProfit

if __name__ == "__main__":
    jobs=[
       {
       "id": "a",
            "deadline": 2,
          "profit": 100
    },
       {
       "id": "b",
            "deadline": 1,
          "profit": 19
    },
      {
       "id": "c",
            "deadline": 2,
          "profit": 27
    },
       {
          "id": "d",
```

```
        "deadline": 1,
            "profit": 25
        },
    {
            "id": "e",
          "deadline": 3,
        "profit": 15
        }
    ]

    n=len(jobs)

    print("Job Schedule Sequence:")
    maxProfit=printScheduling(jobs,n)

    print("\nMaximised Profit: "+str(maxProfit))
```

Output:

```
In [5]: runfile('C:/Users/mayur/OneDrive/Desktop/pyhton program/untitled3.py', wdir='C:/Users/
mayur/OneDrive/Desktop/pyhton program')
Job Schedule Sequence:
a c e
Maximised Profit: 142

In [6]:
```

c) **Complexity of proposed algorithm (Time & Space)**

- Time Complexity: O(N log N), nearly O(N).
- Space Complexity: O(N)

d) **Your comment (How your solution is optimal?)**

The solution is optimal compared to the Brute Force approach and substantially minimizes the complexity. We essentially are working to fill each slot with the best possible time subject to the deadline.

2. To implement Fractional Knapsack problem 3 objects (n=3).
   (w1, w2, w3) = (18, 15, 10)
   (p1, p2, p3) = (25, 24, 15)
   M=20
   With strategy

   A) Largest-profit Strategy

   Ans:

   a) Algorithm: (Pseudocode)
   - We compute a value for each item (based on strategy – largest profit in this case)
   - Obeying a greedy strategy, we take as possible the item with the highest value per pound.
   - If the supply of that element is exhausted and we can still carry more, we take as much as possible of the element with the next value per pound

   b) Code snapshots of implementation

```
# Largets Profit Strategy

class ItemValue:
    def __init__(self, wt, val, ind):
            self.wt = wt
            self.val = val
            self.ind = ind
            self.cost = val//wt

    def __lt__(self, other):
            return (self.cost<other.cost)

class FractionalKnapsack:

    @staticmethod
    def getMaxValue(wt, val, capacity):
            iVal = []
```

```
        for i in range(len(wt)):
            iVal.append(ItemValue(wt[i], val[i], i))

        # sorting items by value
        iVal.sort(reverse=True)

        totalValue=0
        for i in iVal:
            curWt = int(i.wt)
            curVal = int(i.val)
            if capacity - curWt >=0:
                capacity -= curWt
                totalValue += curVal
            else:
                fraction = capacity/curWt
                totalValue += curVal*fraction
                capacity = int(capacity - (curWt*fraction))
                break
        return totalValue

w = [18,15,10]
p = [25,24,15]
M = 20

maxValue = FractionalKnapsack.getMaxValue(w, p, M)
print("Maximum value in Knapsack (Largest Profit Strategy) = ",
maxValue)
```

Output:

```
In [6]: runfile('C:/Users/mayur/OneDrive/Desktop/pyhton program/DAA 6.5.py', wdir='C:/Users/mayur/
OneDrive/Desktop/pyhton program')
Maximum value in Knapsack (Largest Profit Strategy) =  28.2
```

c) Complexity of proposed algorithm (Time & Space)

- Time Complexity: O(N log N)
- Space Complexity: O(N)

d) Your comment (How your solution is optimal?)

The solution is optimal with greedy approach with the value of 28.2. Here the strategy used to compute is very important. We set the strategy of sort in __lt__.

B) Smallest-weight Strategy

Ans:

a) Algorithm: (Pseudocode)

- We compute a value for each item (based on strategy – smallest weight in this case)
- Obeying a greedy strategy, we take as possible the item with the highest value per pound.
- If the supply of that element is exhausted and we can still carry more, we take as much as possible of the element with the next value per pound

b) Code snapshots of implementation

```
# Smallest Weight Strategy

class ItemValue:
    def __init__(self, wt, val, ind):
            self.wt = wt
            self.val = val
            self.ind = ind
            self.cost = val//wt

    def __lt__(self, other):
            return (self.val>other.val)

class FractionalKnapsack:

    @staticmethod
    def getMaxValue(wt, val, capacity):
            iVal = []
            for i in range(len(wt)):
```

```
                iVal.append(ItemValue(wt[i], val[i], i))

        # sorting items by value
        iVal.sort(reverse=True)

        totalValue=0
        for i in iVal:
                curWt = int(i.wt)
                curVal = int(i.val)
                if capacity - curWt >=0:
                        capacity -= curWt
                        totalValue += curVal
                else:
                        fraction = capacity/curWt
                        totalValue += curVal*fraction
                        capacity = int(capacity - (curWt*fraction))
                        break
        return totalValue

w = [18,15,10]
p = [25,24,15]
M = 20

maxValue = FractionalKnapsack.getMaxValue(w, p, M)
print("Maximum value in Knapsack (Smallest Weight Strategy) = ",
maxValue)
```

Output:

```
In [11]: runfile('C:/Users/mayur/OneDrive/Desktop/pyhton program/untitled5.py', wdir='C:/Users/
mayur/OneDrive/Desktop/pyhton program')
Maximum value in Knapsack (Smallest Weight Strategy) =  31.0

In [12]:
```

c) Complexity of proposed algorithm (Time & Space)

- Time Complexity: O(N log N)
- Space Complexity: O(N)

d) Your comment (How your solution is optimal?)

The solution is optimal with greedy approach with the value of 31.0. Here the strategy used to compute is very important. We set the strategy of sort in __lt__.

C) Largest profit-weight ratio strategy

Ans:

a) Algorithm: (Pseudocode)

- We compute a value for each item (based on strategy – smallest weight in this case)
- Obeying a greedy strategy, we take as possible the item with the highest value per pound.
- If the supply of that element is exhausted and we can still carry more, we take as much as possible of the element with the next value per pound

b) Code snapshots of implementation

```
# Largest Profit-Weight Ratio Strategy

class ItemValue:
    def __init__(self, wt, val, ind):
            self.wt = wt
            self.val = val
            self.ind = ind
            self.cost = val//wt

    def __lt__(self, other):
            return ((self.val/self.wt)<(other.val/other.wt))

class FractionalKnapsack:

    @staticmethod
    def getMaxValue(wt, val, capacity):
            iVal = []
```

```
        for i in range(len(wt)):
            iVal.append(ItemValue(wt[i], val[i], i))

        # sorting items by value
        iVal.sort(reverse=True)

        totalValue=0
        for i in iVal:
            curWt = int(i.wt)
            curVal = int(i.val)
            if capacity - curWt >=0:
                capacity -= curWt
                totalValue += curVal
            else:
                fraction = capacity/curWt
                totalValue += curVal*fraction
                capacity = int(capacity - (curWt*fraction))
                break
        return totalValue

w = [18,15,10]
p = [25,24,15]
M = 20

maxValue = FractionalKnapsack.getMaxValue(w, p, M)
print("Maximum value in Knapsack (Largest Profit-Weight Strategy) = ", maxValue)
```

Output:

```
In [12]: runfile('C:/Users/mayur/OneDrive/Desktop/pyhton program/DAA 6.7.py', wdir='C:/Users/mayur/
OneDrive/Desktop/pyhton program')
Maximum value in Knapsack (Largest Profit-Weight Strategy) =  31.5
```

c) Complexity of proposed algorithm (Time & Space)

- Time Complexity: O(N log N)
- Space Complexity: O(N)

d) **Your comment (How your solution is optimal?)**

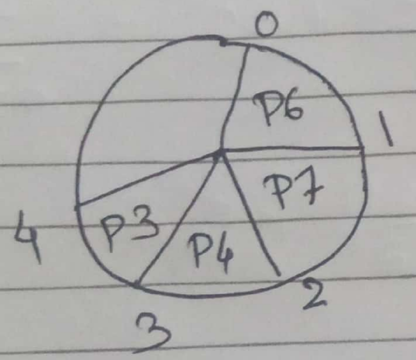
The solution is optimal with greedy approach with the value of 31.5. Here the strategy used to compute is very important. We set the strategy of sort in __lt__.

We find that the approach of profit/weight ratio yields the maximum result and hence is the best of the 3 strategies of Fractional Knapsack.

Q1)
A⇒

P1	P2	P3	P4	P5	P6	P7
3	5	20	18	1	6	30
6	5	2	3	7	4	1
d_1	d_2	d_3	d_4	d_5	d_6	d_7
1	3	$\boxed{4}$	3	2	1	2

max dedline is 4



= P6 + P7 + P4 + P3

= 6 + 30 + 18 + 20

= 74

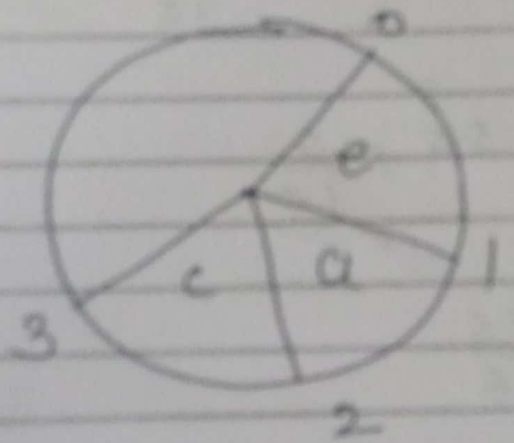
1

(c)

⇒

Job ID	Deadline	Profit		
a	2	100	1	
b	1	19	4	✗
c	2	27	2	
d	1	25	3	✗
e	3	15	5	

max = 3



= c + a + e

= 27 + 100 + 15

= 142

Q2)

⇒ $w_1, w_2, w_3 = 18, 15, 10$

$P_1, P_2, P_3 = 25, 24, 16$

$M = 20$

① Greedy for profit

object	profit	weight
1	25	18
2	24	15

$$\sum_{i=0}^{3} = 25 + \cancel{24} + \frac{2}{\cancel{15}}$$

$$= 25 + \frac{16}{5}$$

$$= 25 + 3.2$$

$$= 28.2$$

② Greedy for weight

Object	Profit	weight
3	15	10
2	$24 \times \frac{10}{15}$	15

$$\sum_{i=0}^{3} = 25 + 24 \times \frac{10}{15}$$

$$= 15 + \frac{80}{5}$$

$$= 15 + 16$$

$$= 31$$

③ $\frac{P}{\omega}$

$= \frac{P1}{\omega 1} = \frac{25}{18} = 1 \cdot 4 \quad x_1 = 0$

$\frac{P2}{\omega 2} = \frac{24}{15} = 1 \cdot 6 = x_2 = 1$

$\frac{P3}{\omega 3} = \frac{15}{10} = 1 \cdot 5 \quad x_3 = \frac{5}{10}$

$$\sum_{i=1}^{n} = 24 + 15 \times \frac{15}{10}$$

$$= 24 + 15 + 0.5$$

$$= \underline{31.5}$$