CSE 431/531: Algorithm Analysis and Design — Fall 2022

Homework 2

Instructor: Shi Li — **Deadline: 10/9/2022**

Name: Amudheswaran Sankaranarayanan

UB name: amudhesw

Problem 1 Construct the Huffman code (i.e, the optimum prefix code) for the alphabet $\{a, b, c, d, e, f, g\}$ with the following frequencies:

Symbols	a	b	c	d	e	f	g
Frequencies	50	20	27	25	29	85	55

Also give the weighted length of the code (i.e, the sum over all symbols the frequency of the symbol times its encoding length).

Sol:

g >> 00

c >> 010

e >> 011

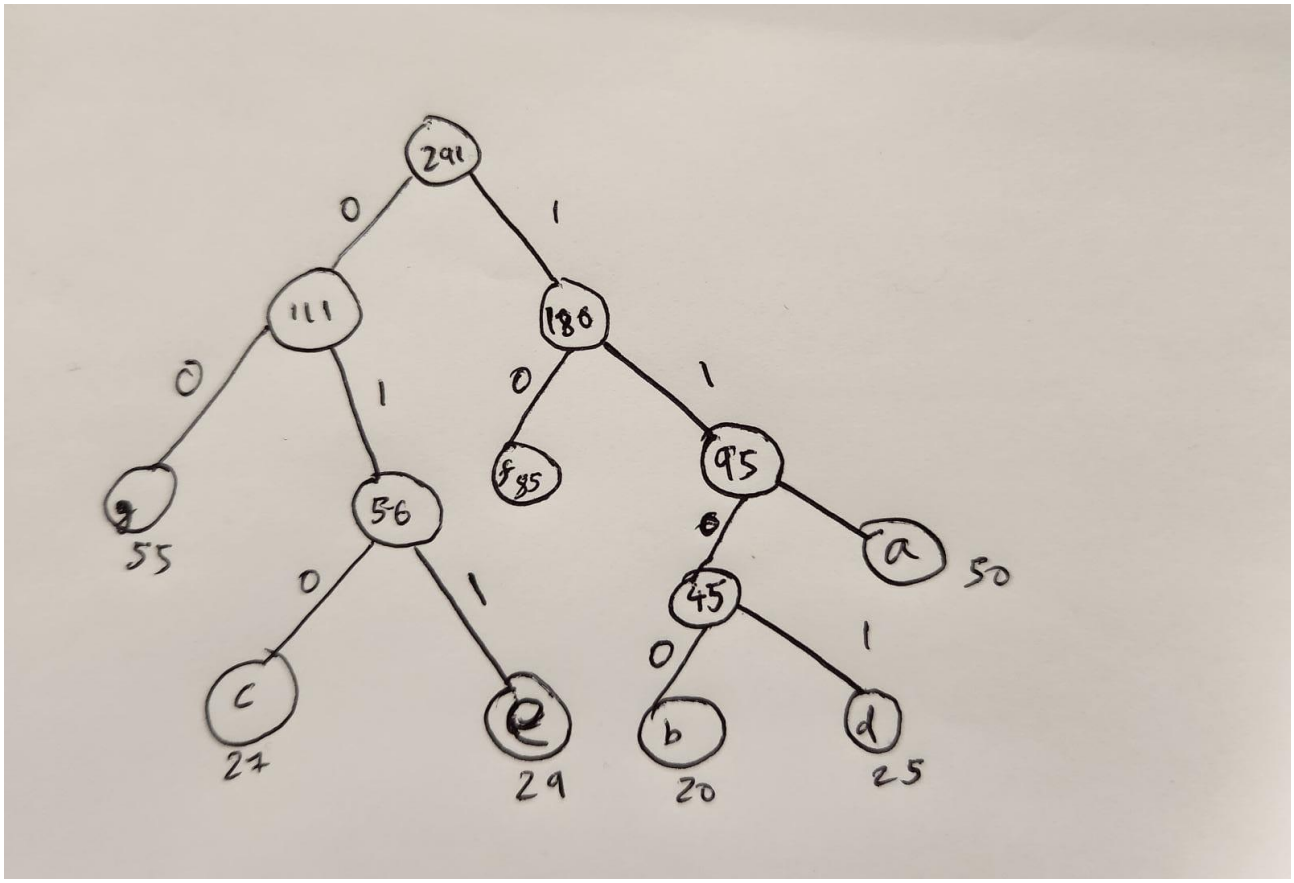
f >> 10

b >> 1100

d >> 1101

a >> 111

Symbol	Frequency	Code	Weight
g	55	00	110
c	27	010	81
e	29	011	87
f	85	10	170
b	20	1100	80
d	25	1101	100
a	50	111	150
Total Weight =			**778**



Problem 2 We are given an array A of length n. For every integer i in $\{1, 2, 3, \cdots, n\}$, let b_i be median of the sub-array $A[1..i]$. (If the sub-array has even length, its the median is defined as the lower of the two medians. That is, if i is even, b_i is the $i/2$-th smallest number in $A[1..i]$.) The goal of the problem is to output $b_1, b_2, b_3, \cdots, b_n$ in $O(n \log n)$ time.

For example, if $n = 10$ and $A = (110, 80, 10, 30, 90, 100, 20, 40, 35, 70)$. Then $b_1 = 110, b_2 = 80, b_3 = 80, b_4 = 30, b_5 = 80, b_6 = 80, b_7 = 80, b_8 = 40, b_9 = 40, b_{10} = 40$.

Hint: use the heap data structure.

Soln:

```python
from heapq import heappush, heappop


iterativeMedianArr = [110,80,10,30,90,100,20,40,35,70]
n = len(iterativeMedianArr)
res = []
minHeap = []  #g
maxHeap = []  #s
for i in range(n):
    #print("##########")
    heappush(maxHeap, -iterativeMedianArr[i])

    heappush(minHeap, -heappop(maxHeap))
    #print("maxHeap after pop >", maxHeap)
    #print("minHeap after push >", minHeap)

    if len(minHeap) > len(maxHeap):
        heappush(maxHeap, -heappop(minHeap))

    if len(maxHeap) != len(minHeap):
        res.append(-maxHeap[0]) #not pop, peek
    else: #even
        #print("even:",min(-maxHeap[0] , minHeap[0]) )
        res.append(min(-maxHeap[0] , minHeap[0]))



print(res)
```

✓ 0.3s

```
[110, 80, 80, 30, 80, 80, 80, 40, 40, 40]
```

This utilizes a maxheap and minheap to split the input stream in two so that the roots are always containing the median.

Run time complexity: O(n) => for loop for entering n elements

*

O(logn) => heap operations

Problem 3 In the interval covering problem, we are given n intervals $(s_1, t_1], (s_2, t_2], \cdots, (s_n, t_n]$ such that $\bigcup_{i\in[n]}(s_i, t_i] = (0, T]$. The goal of the problem is to return a smallest-size set $S \subseteq \{1, 2, 3, \cdots, n\}$ such that $\bigcup_{i\in S}(s_i, t_i] = (0, T]$.

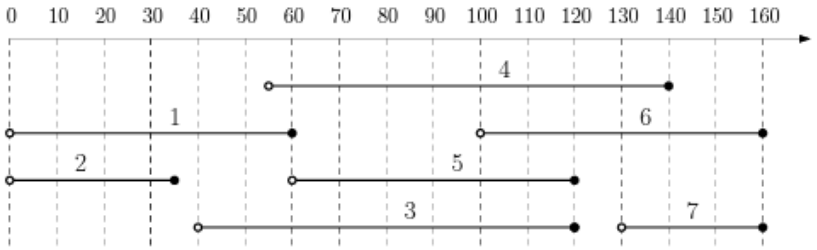


Figure 1: An instance of the interval covering problem.

For example, in the instance given by Figure 1. The intervals are $(0, 60], (0, 35], (40, 120], (55, 140], (60, 120], (100, 160], (130, 160]$. Then we can use 3 intervals indexed by 1, 4, 7 to cover the interval $(0, 160]$. This is optimum since no two intervals can cover $(0, 160]$.

Design a greedy algorithm to solve the problem. it suffices for your algorithm to run in polynomial time. To prove the correctness of the algorithm, it is recommended that you can follow the following steps:

- Design a simple strategy to make a decision for one step.
- Prove that the decision is safe.
- Describe the reduced instance after you made the decision.

Soln:

The strategy here is to pick intervals with larger duration. As per the question, it is evident that the intervals may overlap, and from the ideal solution, we can see that it optimizes by reducing the overlap.

Hence our strategy would be to

1. Maximize job duration
2. Minimize overlap

The decisions are safe as they correctly represent those intervals that will occupy the entire period/time range. If no such job is found, then nothing would be returned.

The reduced instance from the output is

[[0,60] , [55,140] , [130,160]]

This is the optimal path that minimises the no. of intervals in the set.

.

```python
import math

def maxJobs(intervals,start,end):
    intervals.sort()
    resultSet = []
    finish = 0
    intervals.append([math.inf,math.inf]) #padding elemnt
    overlap = math.inf
    # Its The overlap between the next job's start time and current job's finish

    for i in range(len(intervals)):
        if (intervals[i][0] <= start )and ( start - intervals[i][0] <= overlap):
            finish =  max( finish, intervals[i][1] )
            overlap = min(overlap, (start - intervals[i][0] ) )

            if finish == intervals[i][1]:
                prevStart =  intervals[i][0]
        else:

            resultSet.append([prevStart,finish])
            overlap = math.inf # REINITIALIZE TO INF for new job
            start = finish

    if (end > finish ): # jobs dont fully utilize the given interval
        print("jobs dont fully utilize the given interval")
        return None
    return resultSet
intervals = [ [0,60] ,[0,35],[40,120], [ 55, 140] ,[60,120],[100,160],[130,160] ]
start = 0
end =160
print(maxJobs(intervals=intervals,start=start, end=end))
```

[29] ✓ 0.2s

```
[[0, 60], [55, 140], [130, 160]]
```