CSE 431/531: Algorithm Analysis and Design Fall 2021

Homework 2

Instructor: Shi Li **Deadline: 10/14/2021**

Your Name: _____ Your Student ID: _____

Problems	1	2	3	Total
Max. Score	20	30	30	80
Your Score				

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

Symbols	a	b	c	d	e	f	g
Frequencies	50	20	27	25	29	85	55
Encoding	001	0000	100	0001	101	01	11

Weighted length $= 50 \times 3 + 20 \times 4 + 27 \times 3 + 25 \times 4 + 29 \times 3 + 85 \times 2 + 55 \times 2 = 150 + 80 + 81 + 100 + 87 + 170 + 110 = 778$.

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.

We maintain two heaps, one max-heap and one min-heap. In the i-th iteration of our algorithm, the max-heap contains the $\lceil i/2 \rceil$ smallest numbers in $A[1..i]$, and the min-heap contains the $\lfloor i/2 \rfloor$ biggest numbers in $A[1..i]$. Then, the lower median of $A[1..i]$ is the maximum element in the maximum heap. The pseudo-code gives how to maintain the two heaps:

```
1: maxheap ← an empty max-heap
2: minheap ← an empty min-heap
3: for i ← 1 to n do
4:     if i is odd then
5:         if A[i] ≤ minheap.get-min() then
6:             maxheap.add(A[i], A[i])
7:         else
8:             t ← minheap.extract-min(), maxheap.add(t, t), minheap.add(A[i], A[i])
9:     else                                                    ▷ i is even
10:        if A[i] ≥ maxheap.get-max() then
11:            minheap.add(A[i], A[i])
12:        else
13:            t ← maxheap.extract-max(), minheap.add(t, t), maxheap.add(A[i], A[i])
14:    b_i ← maxheap.get-max()
15:    return b
```

In the pseudo-code, extract-min (extract-max) for the min-heap (max-heap) returns and removes the minimum (maximum) element. get-min (get-max) for the min-heap (max-heap) returns but does not remove the minimum (maximum) element. If the min-heap (max-heap) is empty, then get-min (get-max) returns ∞ ($-\infty$). The running time in each iteration of Loop 3 is $O(\log n)$ and thus the overall running time of the whole algorithm is $O(n \log n)$.

Problem 3 A string of "(" and ")" is said to be balanced, if it satisfies the recursive definition:

- The empty string "" is balanced.
- If A is balanced then (A) is balanced.
- If A and B are balanced, then AB is balanced.

For example, "(()())()" is balanced.

In this problem, you are given a string of "(" and ")", the goal is to remove the minimum number of characters so that the residual string is a balanced. For example, if the input is "())((()()))((", then you can remove 3 characters to obtain a balanced string, as follows: "()~~)~~(()())~~)~~~~(~~(".

Design a greedy algorithm to solve the problem. You need to show the algorithm is correct. To get a full score, your algorithm needs to run in $O(n)$ time, where n is the length of the string.

(This is what is posted on Piazza) Given a string A of length n containing only '(' and ')', a valid solution is a matching $M = \{(\ell_1, r_1), (\ell_2, r_2), \cdots, (\ell_k, r_k)\}$ of indices in $[n]$ such that $A[\ell_i] =' ('$, $A[r_i] =')'$, and $\ell_i < r_i$ for every $i \in [k]$. Moreover, for every $i \neq j, i \in [k], j \in [k]$, exactly one of the following two conditions hold:

- $r_i < \ell_j$ or $r_j < \ell_i$. (One pair appears after the other.)
- $\ell_i < \ell_j < r_j < r_i$ or $\ell_j < \ell_i < r_i < r_j$. (The two pairs are nested.)

So, the problem is to find the maximum-size valid matching M.

You prove the following lemma: If for some index $i \in [n]$ we have $A[i] =' ($' and $A[i+1] =')'$, then it is safe to match i and $i+1$. In other words, there is an optimum solution M such that $(i, i+1) \in M$. Take any optimum solution M.

- If $(i, i+1) \in M$, then we are done with the proof of the lemma.
- If neither i nor $i+1$ is matched in M, then adding $(i, i+1)$ to M would give a larger valid matching, contradicting that M is optimum.
- Thus, at least one of i and $i+1$ is matched, but they are not matched to each other. It can not happen that i is matched to some j and $i+1$ is matched some j'. In this case we have $j < i < i+1 < j'$, contradicting the two conditions. So, in M, exactly one of i and $i+1$ is matched.
- If $(i, j) \in M$ for some j, and $i+1$ is not matched in M, then $M \setminus \{(i,j)\} \cup \{(i, i+1)\}$ gives another optimum valid matching. So, we are done.
- If $(j', i+1) \in M$ for some j' and i is not matched in M, then $M \setminus \{(j', i+1)\} \cup \{(i, i+1)\}$ gives another optimum valid matching. We are also done in this case.

Therefore, in every iteration of the algorithm, we find the smallest i with $A[i] =' ($' and $A[i+1] =' ($', and match $A[i]$ and $A[i+1]$. The residual instance is obtained by removing $A[i]$ and $A[i+1]$ from the string A.

To run the algorithm efficiently, for every i from 1 to n, we maintain the number of left parentheses that have not been matched.

```
1: ans ← 0, unmatched ← 0
2: for i ← 1 to n do
3:     if A[i] =' (' then
4:         unmatched ← unmatched + 1
5:     else if unmatched ≥ 1 then                    ▷ A[i] =')'
6:         unmatched ← unmatched − 1, ans ← ans + 1
7: print("we matched ans pairs of parentheses")
```

The running time of the algorithm is $O(n)$.