Ann, Bob, Charlie (*replace with your names*)
COSC 336
3/19/2020

Assignment 5

Instructions.

1. Due date and time: As indicated on Blackboard.
2. This is a team assignment. Work in teams of 3-4 students. Submit on Blackboard one assignment per team, with the names of all students making the team.
3. The exercises will not be graded, but you still need to present your best attempt to solve them. If you do not know how to solve an exercise, say it. This will give me feedback about your understanding of the theoretical concepts.
4. Your programs must be written in Java.
5. Write your programs neatly - imagine yourself grading your program and see if it is easy to read and understand.

   Comment your programs reasonably: there is no need to comment lines like "i++" but do include brief comments describing the main purpose of a specific block of lines.
6. You will submit on **Blackboard** 2 files.

   The **1-st file** is a pdf file (produced ideally with latex and Overleaf) and it will contain the following:

   (a) The solution to the Exercises (see the remark above).
   (b) A short description of your algorithm for the Programming Task, where you explain the dynamic programing approach. Focus on how you have modified MERGE.
   (c) A table with the results your program gives for the data sets indicated for the programming task.
   (d) The java code (so that the grader can make observations) of the program.

   The **2-nd file** is the .java file containing the java source code for Programming Task.

Exercise 15.3-2, textbook, page 439

Prove that a non-full binary tree cannot correspond to an optimal prefix-free code. (note: A non-full binary tree is a binary tree that has a node which has only one child.)

Exercise 15.3-3, textbook, page 439

What is an optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers?

a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21

Can you generalize your answer to find the optimal code when the frequencies are the first Fibonacci numbers?

Programming Task.

The input consists of a sequence of numbers $a[1], a[2,] \dots, a[n]$. Your task is to design an $O(n^2)$ algorithm that finds an increasing subsequence with the maximum possible sum. An increasing subsequence is given by a sequence of indices $1 \leq i_1 < i_2 < \dots < i_k \leq n$ such that $a[i_1] \leq a_[i_2] \leq \dots \leq a_[i_k]$. Note the the indices defining the subsequence are not necessarily consecutive numbers. The program will output the max sum and the increasing subsequence with that sum.

Your algorithm should work in time $O(n^2)$.

For example, for sequence $1, 14, 5, 6, 2, 3$, the increasing subsequence $1, 14$ has the largest sum $1 + 14 = 15$. ($1, 5, 6$ is another increasing subsequence but its sum, $1 + 5 + 6 = 12$ is smaller.)

Input specification: the first line contains n and the second line contains $a_1, \dots, a_n$. Numbers on the same line are separated by spaces. You may assume that n is not bigger than $10,000$ and all the numbers fit in int.

Output specification: the output contains the maximum possible sum of an increasing subsequence.

Sample inputs :
input-5.1.txt
input-5.2.txt
Sample outputs :
answer-5.1.txt
answer-5.2.txt

Test your program on the following inputs:
input-5.3.txt
input-5.4.txt
and report in a table the results you have obtained for these two inputs. Label the results with appropriate text, for example "output for file input-*.*.txt", or something similar.

Hint: You should use a dynamic programming algorithm. For each $i \leq n$, define
$s[i]$ = max sum of an increasing subsequence with last element $a[i]$, and
$p[i]$ = index of the element preceding $a[i]$ in an increasing subsequence with max sum and last element $a[i]$.

These are the "subproblems" used in the dynamic programming approach. As usual we solve them in order: first we compute $(s[1], p[1])$, next $(s[2], p[2])$, and so on till we compute $(s[n], p[n])$. Next we compute $\max\{s[i] \mid i \leq n\}$, which is the value that we seek (the max sum of an increasing subsequence). Using the $p[i]$'s values we can reconstruct the increasing subsequence with this sum, by going backwards. More precisely, the i for which $s[i]$ is maximum is the last index of an increasing subsequence with max sum, and next, starting from this last element, you go from predecessor to predecessor and construct the subsequence.

Now you need to figure out how to solve the "subproblems."

You start with $s[1] = a[1]$ and $p[1] = -1$ (-1 is a just a conventional value that indicates that $a[1]$ has no predecessor), and then, as explained above, in order you compute

$s[2], s[3], \ldots s[n]$ and $p[2], \ldots, p[n]$. You need to think how to compute $s[i]$, using the preceding $s[j]$, $j < i$. At the end, you get the the subsequence in reverse order from last element to the first element.