EE 367 Spring 2018
Homework 8 Total Points = 14
Upload the program files of Problems F and G into laulima

CLRS Problem 16.1-2 (page 422, 1 pt)

CLRS Problem 16.1-4 (page 422, 1 pt)

CLRS Problem 16.2-2 (page 427, 1 pt)

CLRS Problem 14.3-1 (page 353, 1 pt)

CLRS Problem 14.3-4 (page 354, 1 pt). (Hint: Determine how to check if a node's subtree of intervals does not intersect a given interval. It's related to what was discussed in class.)

CLRS Problem 14-1 (page 354, 1 pt per part, 2 pts total)

Problem A (1 pt) [This is a greedy algorithm problem. This is related to CLRS Problem 16.2-4 on page 428.] Given an array of non-negative integers, you are initially positioned at the first index of the array. Each element in the array represents your maximum jump length at that position. Determine if you are able to reach the last index. For example:

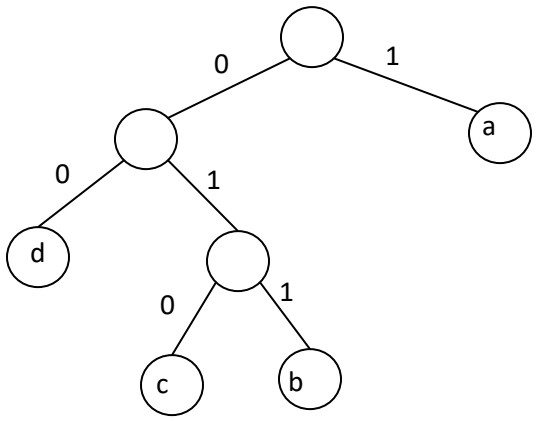
A = [2,3,1,1,4], return true.
A = [3,2,1,0,4], return false.

Write an algorithm to determine whether this is possible or not. Your algorithm should have O(1) space complexity (i.e., amount of space of the algorithm is a constant number of variables) and O(n) time complexity, where n is the number of integers. The indices of the integers are 1, 2, ..., n.

Problem B (1 pt). Draw a Huffman tree for the following symbols and their probabilities.

Symbol	Probability
a	0.1
b	0.05
c	0.6
d	0.1
e	0.025
f	0.025
g	0.05
h	0.05

Problem C (1 pt). Consider the following Huffman tree. The data are ASCII bytes. Using the Huffman tree, encode the string "a b d a c b a"



Problem D (1 pt). Suppose the following string of bits is an encoding using the Huffman tree of Problem 2. Decode the string of bits into a string of characters.

010011100010

Problem E (1 pt). A Huffman code is optimal in the following sense. Suppose you have a set of *n* symbols, where *n* is some integer. The symbols will be denoted by S(1), S(2), ..., S(*n*). Suppose you have a very long stream of data values. Each data value is a randomly generated symbol where the probability that its value is equal to S(*k*) is p(*k*). Assume that the data values are statistically independent of each other.

Now suppose you have a compression algorithm that converts each symbol S(*k*) into a codeword of length L(*k*). Then the average codeword length per data value is $\sum_{k=1}^{n} p(k)L(k)$.

One can prove (though we won't) that among all the lossless compression algorithms, the Huffman code achieves the shortest average codeword length. So it is optimal. However, this is dependent on the data values being statistically independent of each other. If they are statistically dependent then it is not necessarily true. The following illustrates this.

Suppose we have an infinite stream of nibbles (a nibble is four bits). A nibble is equally likely to be either 0000 or 1111. The nibbles are statistically independent. An example of a stream is

0000 1111 0000 0000 1111 0000 1111 1111 ...

Problem F [1 pt]. Recall the longest common substring (LCS) problem from lectures and the CLRS textbook, found in Section 15.4 (pp. 390-397). You are to implement this in a program using dynamic programming with the top-down approach with memoization. Use (modify) the attached program file lcs.c. It's usage is

./a.out <string1> <string2>

It should display the longest common substring of the two strings. (Note that lcs.c has a function to create 2-dimensional arrays, which can be used as memos.) Upload your lcs.c into laulima.

Problem G [1 pt]. Recall the CLRS textbook's Problem 15.2 (page 405), which is to find the longest palindrome subsequence. You are to implement this in a program using dynamic programming either the top-down

approach with memoization or bottom-up (pseudo code for the bottom-up solution is in Homework 7). Use (modify) the attached program file palinsub.c. It's usage is

./a.out <string>

It should display the longest palindrome subsequence of the string. (Note that palinsub.c has a function to create 2-dimensional arrays, which may be useful.) Upload your palinsub.c into laulima.

Problem H [1 pt]. Let's consider encoding a stream of nibbles under the following cases.

Case 1: We will break up the stream into two bit chunks and encode each chunk using a Huffman code. The following is a table of this approach

Symbols	Probability	Huffman codeword
00	0.5	0
11	0.5	1

So the average codeword length is 1. So every 2 bits of data is compressed to 1 bit. So every data bit is compressed to 0.5 bits.

Case 2: We will break up the stream into nibbles and encode each nibble using a Huffman code. The following is a table of this approach

Symbols	Probability	Huffman codeword
0000	0.5	0
1111	0.5	1

So the average codeword length is 1. So every 4 bits of data is compressed to 1 bit, or every data bit is compressed to 0.25 bits. Notice that this is more efficient than case 1 because it takes advantage of statistically dependencies between bits. So the Huffman code is inefficient in Case 1 because of the statistical dependencies of the bits.

Case 3: We will break up a stream into 3-bit chunks and encode each chunk using a Huffman code. To determine the symbols and their probabilities note that every three nibbles A, B, C are four 3-bit chunks:

[A(1), A(2), A(3)] [A(4), B(1), B(2)] [B(3), B(4), C(1)] [C(2), C(3), C(4)]

We'll refer to these four 3-bit chunks as phases 1, 2, 3, and 4. Note that phase 1 and 4 are equally likely to be either 000 or 1111. Phase 2 is equally likely to be either 000, 011, 100, or 111, while phase 3 is equally likely to be 000, 001, 110, or 111. Note that a phase occurs 1/4 of the time in a stream of bits.

Symbols	Probability (or frequency)	freq in phase 1	freq in phase 2	freq in phase 3	freq in phase 4
000	(1/4)(1/2) + (1/4)(1/4) + (1/4)(1/4) + (1/4)(1/2) = 3/8	1/2	1/4	1/4	1/2
111	3/8	1/2	1/4	1/4	1/2
011	(1/4)(1/4) = 1/16	0	1/4	0	0
100	1/16	0	1/4	0	0
001	1/16	0	0	1/4	0
110	1/16	0	0	1/4	0

What is the average codeword length for three data bits assuming the Huffman code? How many compressed bits does a data bit compress to on average?

(Notice that this is worse than Cases 1 or 2 because 3-bit data values look more random than 2-bit data values or 4-bit data values. It doesn't take full advantage of the statistical dependencies of the data bits.)