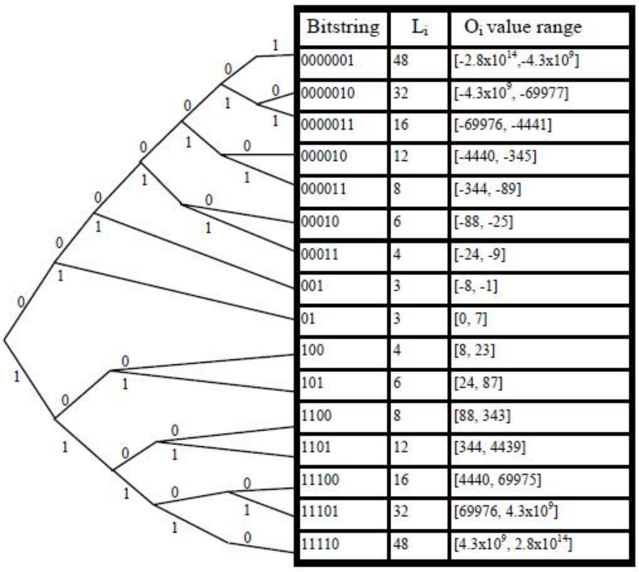
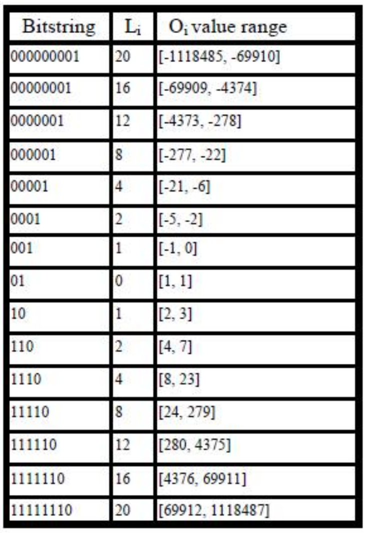
1. How many different Li’s are there in ORDPath? What theory/method is this based on?

   Suggested answer.

   There have two represented type and showed as below:



Bitstring	L_i	O_i value range
0000001	48	$[-2.8\text{x}10^{14}, -4.3\text{x}10^{9}]$
0000010	32	$[-4.3\text{x}10^{9}, -69977]$
0000011	16	[-69976, -4441]
000010	12	[-4440, -345]
000011	8	[-344, -89]
00010	6	[-88, -25]
00011	4	[-24, -9]
001	3	[-8, -1]
01	3	[0, 7]
100	4	[8, 23]
101	6	[24, 87]
1100	8	[88, 343]
1101	12	[344, 4439]
11100	16	[4440, 69975]
11101	32	$[69976, 4.3\text{x}10^{9}]$
11110	48	$[4.3\text{x}10^{9}, 2.8\text{x}10^{14}]$



Bitstring	L_i	O_i value range
000000001	20	[-1118485, -69910]
00000001	16	[-69909, -4374]
0000001	12	[-4373, -278]
000001	8	[-277, -22]
00001	4	[-21, -6]
0001	2	[-5, -2]
001	1	[-1, 0]
01	0	[1, 1]
10	1	[2, 3]
110	2	[4, 7]
1110	4	[8, 23]
11110	8	[24, 279]
111110	12	[280, 4375]
1111110	16	[4376, 69911]
11111110	20	[69912, 1118487]

   The left-hand side figure has 16 code name and the other figure has 15 code name. In the left-hand side figure, all Li values are shown sitting at the leaves of a binary tree. The 0-1 encoding of each Li is determined by the path through the tree from the root to a leaf: a 0 bit issued for each tree edge going up, and 1 bit for each edge going down. A similar binary tree could be constructed for the more regular set of values of Figure 3.2b.

   Deriving each Li bitstring from 0-1 paths through a binary tree clearly provides a prefix free encoding, i.e., no Li bitstring can be a prefix of another Li bit string. We always know when a particular Li bit string ends by following the 0-1 path through the tree until a leaf is reached. At that point the length of the subsequent Oi bit string value is known from the identified Li bit string, so the entire sequence of bits that make up the Li/Oi component pairs of an ORDPATH of Figure 3.1 can be parsed.

2. What is the major advantage of the permutation-based approach when compared with the homomorphic approach? What is the purpose of having the auxiliary data structure(s) in compression?

Suggested answer.

Advantage of permutation-based approach compared with homomorphic approach: By separating structural information from data values, more data duplication can be found by viewing the set of data as a whole. In this way, a higher compression ratio can be achieved.

Having the auxiliary data structure in compression, it is possible to decompress only part of the compressed data rather than doing complete decompressing every single time.