1

1. P(Output = Y|B=Y) = 0.75 P(Output = Y|B =/=Y) = 0.25, and P(output=1|B doesn’t exist) = 0.5. I guess these probabilities are close enough to be differentially private.
2. P(Output=Y|B=Y)=0.5 and P(Output=Y|B=/=Y) = 0.5 (Assuming we just pick a random output or fix B) so would be perfectly differential private? But information loss would be big because no different from flipping a random coin’
3. Sum of list + laplace(1/epsilon) (by community choice)
4. Maybe median? Given on a national scale, removing any one person would shift the median only by 1, but a person with a lot less or a lot more income than the mean could still shift the mean more? (Trillionaire), so the differentially private mean would have to be more noisy
5. Run laplace\_sum(…, epsilon/7) for each summation.

2)

a) Non garbled: Have n \* 2 input wires for each of the input bits, that go into the appropriate boolean gates. The gates forward the output to one of the m output gates, or carry forward/elsewhere if needed.

b) Specification: List the input wires and which gate they go into. For each gate, list the function to compute, and the destination of the output wire. The garbled circuit interpreter can then use this to compute a garbled circuit

c)

1. A dictionary of classes, indexed by gate number. Each gate class contains the garbled gates, the index mapping, the p bits, the gate that their output goes to.
2. A dictionary of keys corresponding to each wire number and 0/1

d) Uhhh (please hope we don’t have to write much code) Pseudo-code attempt:

Keys = {}

For wire in num\_wires:

Keys[wire, 0] = generate\_rsa\_key()

Keys[wire, 1] = generate\_rsa\_key()

Gates = {}

For (gate\_no, output\_gate, func, wire\_A, wire\_B, wire\_C) in initial\_gate\_info: (assume this holds info we need for gate generation)

table = {}

P\_A = random(0, 1) (assume PRNG like a wall of lava lamps or something)

P\_B = random(0, 1)

P\_C = random(0, 1)

For i in [0, 1]:

For j in [0, 1]:

X = i XOR P\_A

Y = j XOR P\_B

Z = functions[func](X, Y)

Z’ = Z XOR P\_C

Table[I, j] = encrypt(encrypt((Keys[wire\_C, Z], Z’), Keys[wire\_A, X]), Keys[wire\_B, Y])

Gates[gate\_no] = Gate\_class(gate\_no, tables, output\_gate)

e) Initially:

Alice->Bob: Alice’s input key k[alice\_input\_wire, alice\_input\_value] for each input bit

Alice -> Bob: The garbled tables and decryption mapping for the final table

Then bob receives his input key

Alice -> Bob: 2 public keys

Bob -> Alice: Symmetric key encrypted with one of the public keys

Alice -> Bob: m1, m2 encrypted with either bob’s key, or the wrong key

So that bob receives k[bob\_input\_wire, bob\_value]

Bob calculates the table output

Bob -> Alice: Final circuit outcome

f) decrypt\_gate(table, p1, p2, k1, k2):

val = table[p1, p2] (Use p-bits to index the garbled table)

val = decrypt(val, k1) (Decrypt with k[w1, alice\_value])

val = decrypt(val, k2) (Decrypt with k[w2, bob\_value])

return val (this is k[output, output\_value])

3

A)

The biggest value of k is two (male, 62, hamburg)

The biggest l is 1 (male, 62, hamburg) contains only people with gastritis

B)

Age would have the biggest effect on k, leaving only two e-classes (male, hamburg) and (female, berlin) which both contain 4 people

The l value would also increase to 3 as both classes have people with 3 distinct diseases

C)

The input to MD5 for me was LKNMSDXX for some salt, by trying all the salt in succession we can find the one that results in my pseudo\_id

There are 90 possible salts (9 \* 10) so we could end up calling the hash function 90 times. This would take 90\* 10^-6 = 90 microseconds

Once we have found the salt we can create a lookup table by enumerating and hashing the whole input space.

D) Kossai:

This is my reasoning for this question (question 3)d) ):

We have the NI number being LKNMSD

We have essentially two cases in terms of the positions of the two digits of the salt

1) The two digits are next to each other (together). That means that they can be placed at 7 different positions in the string (see below). Each \_ represents a position for a digit of the salt. Each block of \_ \_ hence represents 1 position at which both digits get appended to the string.

\_ \_ L \_ \_ K \_ \_ N \_ \_ M \_ \_ S \_ \_ D \_ \_

2) The two digits are separate (not together). See below. Each \_ represents a position for a digit of the salt. That therefore means that they can be placed at 7 choose 2 positions. Remember in that case the two digits are not placed next to each other AND also they cannot be at a same position (i.e. a \_).

\_ L \_ K \_ N \_ M \_ S \_ D \_

That means that they can be placed at 7 choose 2 positions. 7 choose 2 gives us 21.

Combining both cases detailed above gives a total number of positions of 21 + 7 = 28.

Moreover, we know from the previous question that there are 9\*10 salts possible per position. That gives us a total number of combinations/possibilities of 90 \* 28 = 2520.

Finally, in this question, each salted ID is hashed twice, that means two hash operations per salted ID.

Therefore, this would take a computational time of 2520 \* 2 \* 10^-6 = 5.04 \* 10^-3 seconds = 5.04 milliseconds.

Let me know if you guys agree with me!! Anyone?? Sounds reasonable