General clarifications / advice:

- getNgramLogProbability() should compute conditional probability of the final word in the n-gram given the preceding words. “From” should be inclusive and “to” should be exclusive. If the n-gram is longer than the order of your model, you can still compute the probability; think of what Markov assumption tells you.
- Make sure you are prepending *START* and appending *STOP* symbols to every sentence; see how it’s done in EmpiricalUnigramLanguageModel.
- We will run a sanity check on your code as described in the assignment. Remember that you are only given 50M of memory for this part, so don’t initialize your hashmaps to a huge initial size, or else you’ll get an Out Of Memory error in sanity check.

ArrayIndexOutOfBounds exception:

- -1 in the decoder, you might be either handling unseen words incorrectly or dividing by 0 somewhere in your computation of alpha.

Debugging failed spot checks (context distributions not summing to 1):

- Make sure your code matches the formulas in your derivation. For example, very frequently the issue is caused by incorrect computation of alpha values (adding back more probability mass than you subtracted with the discounting factor).
- If your sums are substantially >1, your unseen unigram probability may be too large. Try returning $$1 / |V|$$ or another small constant for those. Do not return -Inf.
- You may be losing precision because of repeating exp() and log() operations too many times. Try to simplify your calculation (for example, don't call getNgramLogProbability() recursively).
- If you use bit twiddling in your hash function, don’t forget to cast the numbers to long before calling bit operators. If you shift a 32-bit int by 40 bits, information gets lost and different n-grams end up with the same ID.

My model takes too much time to build / uses too much memory:

- Calculate (approximately or accurately) the memory usage in your implementation. Java objects (including array and wrapper objects like Integer and Double) cost 20 bytes, while primitives cost 4 bytes. Make sure that the result of your calculation is under the given memory limit.

- We suggest you build your own hashmaps instead of using the provided TIntOpenHashMap, so that you can use Primitives rather than Objects for keys and avoid overhead.
- Avoid looping over the whole vocabulary at test time, instead you can precompute all necessary counts in training. Try avoiding nested loops.
- Think of your how you set your load factor and initial hash table size. Frequent resizing or overly high load factor might slow down your implementation.
- In some cases, counts can be stored as plain arrays instead of hashmaps; see if you are possibly building unnecessary hashmaps.
- Check if your collision rate is too high. You should expect the % of collisions to be roughly equivalent to your load factor. If they are too frequent, you may need a new hash function.
- Your hash function itself may be too slow to compute. Try to think what could be slowing it down (for example, try reformulating everything as bit operations).
- Check if you are overusing log(), exp() and abs() in your code.

Low BLEU score:

- This could be attributed to two reasons:
    - For unseen words, you should assign a constant unigram probability. We recommend using 1/|V|. You might be using a higher value than this.
    - Value of the discount factor "d" you are using is probably not tuned properly. You should experiment with different values of d. You could also add this analysis in your write-up.
- If these two changes don't work, you probably have a bug in your code. Since you're passing the spot checks, your counts are probably right, so you should make sure that you're computing alphas correctly, backing off appropriately, etc. I recommend going through your code in very fine detail and comparing to the notes from Chan's recitation to make sure that you're implementing exactly what is described there. A few things to check:
    - That you used discounted context fertility when backing off, and only when backing off
    - That you use discounted raw counts when calculating trigram probabilities
    - That the context fertility was calculated correctly (when they are incorrect, but consistently incorrect, you can still pass spot checks, so still check that the context fertility counts are correct)