Problem 2

The following steps show how to find top-100 most frequent bigrams from the three given text files:
1. Concatenate the text from files 001.txt 002.txt 003.txt into a single file using the Unix 'cat' command.

```
\$ cat 001.txt 002.txt 003.txt > combined.txt
# no output to console, just a new file with the combined text
```

2. Next, I use a one-line command to print the top 100 most frequent bigrams. Below I break down the components of this one line command.

```
\$ tr -sc "[A-Z][a-z][0-9]'" '[\012*]' < combined.txt | \
  tr '[A-Z]' '[a-z]' | \
  awk -- 'prev!="" { print prev,$0; } { prev=$0; }' | \
  sort | uniq -c | sort -nr | \
  head -n200

output:
9 in the
6 for the
5 i was
4 the race
4 of the
```

The first call of the tr command replaces any non-word character with a newline character using the -c (complement) parameter. The -s (squeeze) option specifies to replace a sequence of or more matches with only a single occurrence of the replacement character. This accounts for any empty lines that may occur due to multiple punctuations. The result of this command is a list of words separated by newlines and no punctuations.
The second call of tr changes all words to lowercase. The awk command prints out the previous and current word in one line. The second awk statement places the previous word on the next new line. The result is then piped to the sort and uniq -c which counts the occurrences of each bigram. Sort is called first to avoid any duplicate counts. The sort -nr sorts the bigram count in descending order. The head -n100 prints the first 100 lines (the 100 most frequent bigrams).

Problem 3

Although smaller n-grams don't capture long-range dependency really well, a model with larger n-grams could theoretically do so. But in practice, there is never enough data to train such large n-grams. i.e. A 200-gram model would require a very large training set, which may not be feasible in a lot of cases.

Problem 4

4.1

$P(A) = 1/3$
$P(B) = 1/3$
$P(N) = 1/3$

Problem 4 continued on next page. . .

4.2

$P(A|A) = 1/3$
$P(A|B) = 1/3$
$P(A|N) = 1/6$
$P(B|A) = 1/3$
$P(B|B) = 1/3$
$P(B|N) = 1/3$
$P(N|A) = 1/3$
$P(N|B) = 1/3$
$P(N|N) = 1/3$

4.3

1. $PP(w_1 w_2 w_3 ... w_n)^{-\frac{1}{N}}$; $PP(\text{ABANABB})^{-\frac{1}{7}} = 3$

2. $PP(w_1 w_2 w_3 ... w_n)^{-\frac{1}{N}}$; $PP(\text{ABANABB\#})^{-\frac{1}{8}} = 0$

4.4

1. $linear\_smoothing(PP(\text{ABANABB})^{-\frac{1}{7}}) = 3$

2. $linear\_smoothing(PP(\text{ABANABB\#})^{-\frac{1}{8}}) = 3.46$

Problem 5

In simple linear interpolation, three scalar values help mix the unigram, bigram, and trigram models. The $\lambda_1, \lambda_2, \lambda_3$ from each model will ultimately sum to 1. These values are normally set by a held out dataset separate from the training set, so that the parameters can be tuned to maximize the probability of the heldout data. If we were to tune these λ s on the entire training set from which the unigram, bigram, trigrams counts were computed from instead, the values will reflect the distribution of the training set, but may not generalize well for an unknown set, making the interpolation not useful for new data. The λs will most likely have equal weights = 1/3.
If I only had access to the training set, I would still use cross-validation by splitting the training into two sets. I think this would be the best approach.

Problem 6

Without smoothing for the Bigram LM, we tended to underestimate the occurrence of OOV during training. We could possibly account for this by creating a new token $< Unkw >$ and adding it to the training set. During the text normalization phase, we can change any word not in the original lexicon into $< Unkw >$. And we can train for its probabilities as we would for any other word. When decoding, use the $< Unkw >$ token probability for words not seen in training.