This assignment is due at 16:00 GMT on 9 Feb, 2017. Late submissions will receive a mark of zero, under the Informatics late policy (http://web.inf.ed.ac.uk/infweb/student-services/ito/admin/coursework-projects/late-coursework-extension-requests). An assignment submitted at 16:01 GMT on 9 Feb is considered late. Before you start, please read this page carefully. Submissions that do not follow the Ground Rules (at the bottom of the page) will receive a mark of zero.

Decoding | Coursework 1

Decoding is the process of taking an input sentence in a source language (e.g. French):

honorables sénateurs , que se est - il passé ici , mardi dernier ?

…And finding its best translation into the target language (e.g. English) under your model:

honourable senators , what happened here last Tuesday ?

To decode, we need to model the probability of an English sentence conditioned on the French sentence. You did some of the work of creating a simple model of this kind in Lab 1. In this assignment, we will give you some French sentences and a probabilistic model consisting of a phrase-based translation model $p_{\text{TM}}(\mathbf{f}, \mathbf{a} \mid \mathbf{e})$ and an n-gram language model $p_{\text{LM}}(\mathbf{e})$. We will assume a noisy channel decomposition and our goal will be to implement the following *decision function*.

$$\begin{aligned}
\mathbf{e}^* &= \arg\max_{\mathbf{e}} p(\mathbf{e} \mid \mathbf{f}) \\
&= \arg\max_{\mathbf{e}} \frac{p_{\text{TM}}(\mathbf{f} \mid \mathbf{e}) \times p_{\text{LM}}(\mathbf{e})}{p(\mathbf{f})} \\
&= \arg\max_{\mathbf{e}} p_{\text{TM}}(\mathbf{f} \mid \mathbf{e}) \times p_{\text{LM}}(\mathbf{e}) \\
&= \arg\max_{\mathbf{e}} \sum_{\mathbf{a}} p_{\text{TM}}(\mathbf{f}, \mathbf{a} \mid \mathbf{e}) \times p_{\text{LM}}(\mathbf{e})
\end{aligned}$$

Since multiplying many small probabilities together can lead to numerical underflow, we'll work in logspace using the equivalent decision function:

$$\mathbf{e}^* = \arg\max_{\mathbf{e}} \log \left(\sum_{\mathbf{a}} p_{\text{TM}}(\mathbf{f}, \mathbf{a} \mid \mathbf{e}) \times p_{\text{LM}}(\mathbf{e}) \right)$$

We will keep the model fixed for this assignment. Your challenge will be to write a program that, given an input French sentence, attempts to find its most probable English translation under this model using this decision function.

Part 1: Getting Started [10 marks]

You will need two things for this assignment. The first is an answer sheet on which you are required to submit your answers to the inline questions below. Follow this link (https://www.overleaf.com/latex/templates/infr1113-homework-1-template/bkqsppbgkqnn) and open the

template to get started.

The second is code and data. At the dice commandline, clone the coursework repository:

```
git clone https://github.com/INFR11133/hw1.git
```

You now have simple decoder and some data files. Take a look at the input French sentences, from a speech in the Canadian parliament:

```
cat data/input
```

Let's translate them!

```
python2.7 decode | tee output.default
```

The `decode` program translates the contents of `data/input` and writes the result to the terminal and to the file `output`. It translates using a *phrase-based* translation model: a model that replaces sequences of French words—called *phrases*—with sequences of English words.[1] The replacement is one-to-one and every input word must be accounted for by exactly one phrase, but the phrases can be reordered. This means that to define the probability of a translation, our model requires probabilities for the choice of segmentation into phrases, the translation of the phrases, and the permutation of the target phrases. However, we will make the simplifying assumption that segmentation and ordering probabilities are uniform across all possible choices, so our translation model will only assign different probabilities to translations that use different phrases to translate the input; the ordering of those phrases is only influenced by an n-gram language model. This means that $p(\mathbf{e}, \mathbf{a} \mid \mathbf{f})$ is proportional to the product of the n-gram probabilities in $p_{\text{LM}}(\mathbf{e})$ and the phrase translation probabilities in $p_{\text{TM}}(\mathbf{f}, \mathbf{a} \mid \mathbf{e})$. In phrase-based translation, the output sequence can be permuted, but the default decoder does not do this.[2] We'll discuss the model and the decoding algorithm in more detail below.

You can probably get the gist of the Canadian senator's complaint from this translation, but it isn't very readable. There are a couple of possible explanations for this:

1. Our model of $p(\mathbf{e} \mid \mathbf{f})$ gives high probability to bad translations. This is called *model error*.
2. Our model is good but our decoder fails to find $\arg\max_{\mathbf{e}} p(\mathbf{e} \mid \mathbf{f})$. This is called *search error*.

How can we tell which problem is the culprit?[3] One way would be to solve problem 2, a purely computational problem. If we do and the resulting translations are good, then our work is finished. If the resulting translations are still poor, then we at least know that we should focus on creating better models. For this assignment, we will focus only on problem 2. (But don't worry, we'll revisit problem 1 repeatedly throughout the course.)

If we're going to improve search so that the decoder returns translations with higher probability, we need some way to measure how well it's doing. You can compute a value that is proportional to $p(\mathbf{e} \mid \mathbf{f})$ using `compute-model-score`.

```
python2.7 compute-model-score < output.default
```

It will take a few minutes. Make a cup of tea.

The `compute-model-score` program computes the probability of the decoder's output according to the model. It does this by summing the probabilities of all possible alignments that the model could have used to generate the English from the French, including translations that permute the phrases. That is, givne a pair $\mathbf{f}$ and $\mathbf{e}$ it exactly computes:

$$\log\left(\sum_{\mathbf{a}} p_{\text{TM}}(\mathbf{f}, \mathbf{a} \mid \mathbf{e}) \times p_{\text{LM}}(\mathbf{e})\right)$$

This value is proportional to $p(\mathbf{e} \mid \mathbf{f})$ up to a constant $\frac{1}{p(\mathbf{f})}$. In general it is intractable to compute this sum, and if you inspect the code you'll find that the implementation uses an algorithm whose worst-case runtime is exponential in the length of the input sentence, and indeed this computation is provably NP-hard. But exponential worst case is an upper bound, and for these particular instances the sum will only take a few minutes. It is easier to do this than it is to find the optimal translation.

I highly recommend that you look at the code (https://github.com/INFR11133/hw1/blob/master/compute-model-score) for `compute-model-score`. It uses an unpruned exact dynamic program to compute the sum, so it may give you some hints about how to do the assignment! It also contains some useful utility functions to add probabilities in logarithmic space (https://github.com/INFR11133/hw1/blob/master/compute-model-score#L16) and manipulate bitmaps (https://github.com/INFR11133/hw1/master/compute-model-score#L8).

A tour of the default decoding algorithm

Now let's turn our attention to the decoder you've been given. It generates the most probable translations that it can find, but it uses three common approximations that can cause search error.

The Viterbi approximation

The first approximation of our decoder is the *Viterbi approximation*. Instead of computing the intractable sum over all alignments for each sentence, it seeks the best single alignment and uses the corresponding translation.

$$\begin{aligned}\mathbf{e}^* &= \arg\max_{\mathbf{e}} \log\left(\sum_{\mathbf{a}} p_{\text{TM}}(\mathbf{f}, \mathbf{a} \mid \mathbf{e}) \times p_{\text{LM}}(\mathbf{e})\right) \\ &\approx \arg\max_{\mathbf{e}} \log\left(\max_{\mathbf{a}} p_{\text{TM}}(\mathbf{f}, \mathbf{a} \mid \mathbf{e}) \times p_{\text{LM}}(\mathbf{e})\right)\end{aligned}$$

This approximation vastly simplifies the search problem, since as we've already discussed, computing the sum for even a single $\mathbf{e}$ can take time exponential in sentence length. This is why we require a separate script `compute-model-score` to compute $p(\mathbf{e} \mid \mathbf{f})$, rather than having the decoder itself do this. Computing sums for an exponential number of outputs is doubly exponential (that's bad).

Translation without reordering

The second approximation of our decoder is that it translates French phrases into English without changing their order. That is, it has a heuristic *distortion limit* of zero. So, it only reorders words if the reordering has been memorized as a phrase pair. For example, in the first sentence, we see that *mardi dernier* is correctly translated as *last Tuesday*.

```
head -1 data/input
head -1 output.default
```

If we consult `data/tm`, we will find that this happens because the model has memorized the phrase translation `mardi dernier ||| last Tuesday`.

```
grep "mardi dernier" data/tm
```

But in the second sentence, we see that *Comité de sélection* is translated as *committee selection*, rather than the more fluent *selection committee*. To show that this is a search problem rather than a modeling problem, we can generate the latter output by hand and check that the model really prefers it.

```
head -2 data/input | tail -1 > example
python2.7 decode -i example | python2.7 compute-model-score -i example
echo a selection committee was achievement . | python2.7 compute-model-score -i e
xample
```

The scores are reported as log-probabilities, and higher scores (with lower absolute value) are better. We see that the model prefers *selection committee*, but the decoder does not consider this word order since it has never memorized this phrase pair. Moreover, you can see from the language model and translation model scores of this example that it is the language model that correctly selects the correct ordering. Since we have only changed the order of the words, the translation model score is identical for both translations. So, if our decoder could search over more permutations of the translated phrases, the language model would allow it to choose more fluent sentences.

Dynamic programming

Not searching over permutations of the translated phrases is harmful because the decoder cannot find translations with higher model score, but it is beneficial in one way, because it admits a straightforward dynamic program, which will now define under the simplifying assumption of a bigram language model (the default decoder uses a trigram language model). Let $\mathbf{f} = f_1 \dots f_I$ and, for each phrase $f_i \dots f_j$ let $t(f_i \dots f_j)$ denote its set of possible phrase translations. The problem our decoder must solve is: what is the most probable translation under our model, under the constraint that phrase translations are one-to-one, in the same order as the source, and covering all source words exactly once?

We called this translation $\mathbf{e}^*$ above, so let's continue to use that notation. Although $\mathbf{e}^*$ must be chosen from a large set of translations, each of which can be decomposed into a smaller set of individual decisions that factor with $p_{\text{TM}}(\mathbf{f}, \mathbf{a} \mid \mathbf{e})$ and $p_{\text{LM}}(\mathbf{e})$. Since our language model must define a stop probability, we know that the best translation must contain a bigram probability of the $STOP$ symbol, conditioned on its final word. Let's use $h(j, e)$ to denote the highest probability sequence of English words ending in word e that translates $f_1 \dots f_j$, and $p(h(j, e))$ denote the probability of this partial translation: that is, the product of the translation model probability for all phrases used and the language model probability for the entire sequence up to word e. Given this definition, we can define $\mathbf{e}^*$ as the product of two terms:

$$\mathbf{e}^* = \arg\max_{h(I,e)} \log p(h(I, e)) + \log p_{\text{LM}}(STOP \mid e)$$

This is a good start, because now we've defined $\mathbf{e}^*$ as a choice from at most V_E objects (each corresponding to the choice of e in $h(I, e)$) rather than an exponential number of objects (as a practical matter, we'll only need to inspect $h(I, e)$ for those e that can actually appear in a valid translation of $\mathbf{f}$, a much smaller set than V_E). But now we have a new problem: how do we define $h(I, e)$? To answer this, we'll actually answer a more general question: how would we define $h(j, e)$ for any j between 1 and I? Since it is constructed from a sequence of phrase translations, let's break it into two parts: a prefix that

translates the words $f_1 \dots f_i$ and the final English phrase, which must be a translation of the French words $f_{i+1} \dots f_j$ for some position $0 \leq i < j$. There are many possible choices for i, the translation of $f_1 \dots f_i$ may end in many possible English words, and there may be many translations of $f_{i+1} \dots f_j$. We must maximize over all combinations of these choices:

$$\begin{aligned} h(j, e) = \arg \max_{h(i,e')e_1 \dots e_k e} & \log p(h(i, e')) + \log p_{\text{TM}}(f_{i+1} \dots f_j \mid e_1 \dots e_k e) + \\ & \log p_{\text{LM}}(e_1 \mid e') + \sum_{k'=1}^{k-1} \log p_{\text{LM}}(e_{k'+1} \mid e_{k'}) + \\ & \log p_{\text{LM}}(e \mid e_k) \end{aligned}$$

Don't be alarmed by the notation: all this says is that $h(j, e)$ must consist of the best choice of a pair $h(i, e')$ and phrase $e_1 \dots e_k e$, where the final word of the phrase must be e. The first term is the probability of $h(i, e')$, the second is the probability of translating $f_{i+1} \dots f_j$ by $e_1 \dots e_k e$, and the remaining terms compute the language model probability of the new English words by iterating over them. All that is left is to define a base case, using ϵ to denote an empty string:

$$h(0, e) = \begin{cases} \epsilon & \text{if } e = START \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note that this implies:

$$p(h(0, e)) = \begin{cases} 1 & \text{if } e = START \\ 0 & \text{otherwise} \end{cases}$$

Using induction, convince yourself that this recursion defines every highest-probability *hypothesis* (or partial translation) $h(j, e)$ for every choice of j and e, because it recursively considers all possible ways of reaching that hypothesis. By extension, it must also define the highest-probability translation $\mathbf{e}^*$.

To implement this dynamic program, we actually compute the recursion left-to-right, from smaller to larger values of j. In this way, we always have all the information we need when computing $h(j, e)$. As a practical matter, the way we do this is to ask, for each $h(i, e')$, which larger hypotheses $h(j, e)$ it might be a maximizing prefix to, and then for each one compute the probability of $h(j, e)$ as if this were true. If the newly computed probability is indeed higher than any probability we previously computed for $h(j, e)$, we store it with $h(j, e)$ and make $h(i, e')$ its predecessor. The MT jargon term for this is *recombination*.

The form of the recursion gives us a strong hint about the upper bound complexity of the dynamic program, since if we consider all possible assignments of i, j, e, e', and $e_1, \dots, e_k$ in the central recursion, we can deduce that complexity is $\mathcal{O}(I^2)$ if phrases can be arbitrarily long, and $\mathcal{O}(IK)$ if phrases have a maximum length K (which is true in the default decoder and most practical implementations). However there is a large factor (constant in the input length) due to language model computations.

Beam search, aka stack decoding

The third approximation of our decoder is pruning: as it constructs the dynamic program from left to right, for each source position j it remembers only the highest probability values for $h(j, e)$ (over all $e \in V_E$) and discards the rest. The decoder uses *histogram pruning*, in which at most s hypotheses are retained at each position j. By default it only keeps 1 hypothesis. Pruning introduces approximation into

the search, because a hypothesis leading to the overall best translation may be pruned during search. At each position j, we keep the s best hypotheses in a *stack* (the unfortunate MT jargon term for a priority queue).

The default decoder also implements another common form of pruning, which is that it only considers the k most probable translations for each phrase. By default, $k = 1$.

This approach to pruning, in which we iterate through a set of successively longer partial translations, considering only a fixed number at each time step, is also known as *beam search*.

Putting it all together

The code consists of two distinct sections: one defines the dynamic programming recursion, starting at line 42 (https://github.com/INFR11133/hw1/blob/master/decode#L42).

- Line 42 simply defines objects of the form $h(i, e)$, here called `state` objects.
- The function `initial_state` at Line 45 defines the initial state $h(0, START)$.
- The function `assign_stack` at Line 49 takes a state as input, and returns the index of the stack it should appear in.
- The function `extend_state` at line 58 computes all possible extensions of a state `s` with reference to input sentence `f`.

For the first part of the homework below (and probably the second), these functions are the only part of the code that you need to modify!

A generic stack decoding algorithm starting at line 88 (https://github.com/INFR11133/hw1/blob/master/decode#L88) closely follows the pseudocode in Koehn's textbook (Figure 6.6 on p. 165).

- Lines 91–94 define data structures: we have a hypothesis data structure that summarizes a partial English translation of the French sentence parameterized by the state objects defined above, but also including information about the best previous state. There is also a set of n+1 stacks. `stack[i]` will eventually hold different hypotheses about ways to translate exactly i words of the French. The hypotheses in a stack are indexed by their state, so that each state can appear at most once in a stack.
- Line 95 places an initial hypothesis in the initial stack (Fig 6.6 line 1).
- At line 96, we iterate over the stacks, processing each in turn. In other words, we process the hypotheses in order of the number of French words they have translated. (Fig 6.6 line 2).
- At line 97, we iterate over all hypotheses in the stack (Fig 6.6 line 3). Prior to this, we sort the hypotheses according the LM score and then prune the stack, leaving only the top s hypotheses (Fig 6.6 line 9).
- Lines 98-103 iterate over all possible extensions of the hypothesis (Fig 6.6 line 4-6).
- Line 104 adds the new hypothesis to a stack (Fig 6.6 line 7). Notice that in the second condition of line 49 we also do recombination: if there was already a hypothesis with the same LM state but lower probability, we replace it with the newly created one, which represents a higher-probability path with identical completions (Fig 6.6 line 8).
- Lines 106–109 print the best hypothesis. First, we choose the best hypothesis in the final stack, which is simply the one with the highest probability (line 51). Then we trace back through the sequence of hypotheses that lead to it, and print out the corresponding English words in order using the recursive function `extract_english` (this is not a closure. I just included it at this point to keep all of the logic in a natural order).

It's quite possible to complete the assignment without modifying any part of the stack decoding algorithm!

Now that we've seen how all of this works, it's time to experiment with some of the pruning parameters, and see how they trade search accuracy for speed.

```
time python2.7 decode > output.default
python2.7 compute-model-score < output.default
time python2.7 decode -s 100 -k 10 > output.s=100.k=10
python2.7 compute-model-score < output.s=100.k=10
vimdiff output.default output.s=100.k=10
```

(Type `:q:q` to exit vimdiff).

Question 1 [10 marks]: Experiment with different combinations of values for the stack size parameter `-s` and the maximum number of translations `-k`, until you can no longer obtain any improvements.

1. [4 marks] How do changes in these parameters affect the resulting log-probabilities?
2. [2 marks] How do they affect speed?
3. [2 marks] How do they affect the translations? Do the translations change? Do they make more sense? Are they more fluent? Give examples.
4. [2 marks] What is the maximum log-probability you can obtain?

Part2: Local Reordering [50 marks]

Your task is to **find the most probable English translation**.

Remember that our model assumes that any segmentation of the French sentence into phrases followed by a one-for-one substitution and permutation of those phrases is a valid translation. The baseline approach that I want you to explore in this part of the assignment is one that considers only a very limited space of permutations: the ability to swap translations of adjacent phrases. More precisely, if $\vec{e}_1$ is the translation of French words $f_i \dots f_k$ and $\vec{e}_2$ is the translation of French words $f_{k+1} \dots f_j$, then your output can contain them in the order $\vec{e}_2 \vec{e}_1$. Notice that since phrases can be swapped only if they are translations of adjacent phrases, further swaps involving $\vec{e}_1$ and $\vec{e}_2$ are no longer possible, because the new right neighbour of $\vec{e}_1$ was not adjacent to it in the original order, likewise the left neighbour of $\vec{e}_2$. In other words, we cannot permute sequences of three or more phrases under this definition of reordering.

To be very precise about what the new space of possible translations looks like, let's examine a worked example.

```
echo un Comité de sélection > example
```

To keep things simple, we'll only consider the case where there are at most two translations per phrase. You can control this using the -k parameter to the decoder. To see the phrases that will actually be used, insert the following line just below line 61 (https://github.com/INFR11133/hw1/blob/master/decode#L61):

```
sys.stderr.write("%s ||| %s ||| %f\n" % (" ".join(f[s.i:j]), phrase.english, phra
se.logprob))
```

Now run the decoder:

```
python decode -k 2 -i example
```

You should see the following phrase table on stderr:

```
un ||| a ||| -0.128525
un ||| an ||| -0.912392
Comité ||| committee ||| -0.277924
Comité ||| Committee ||| -0.330485
Comité de ||| committee ||| -0.511883
Comité de ||| Committee on ||| -0.511883
de ||| of ||| -0.313224
de ||| to ||| -0.746520
de sélection ||| for determining qualified ||| 0.000000
sélection ||| selection ||| -0.054358
sélection ||| determining qualified ||| -0.929419
```

How many translations can we create from the above phrase table? Our model consists of three steps: segmentation, permutation, and substitution. (Note that under the assumptions of our model, permutation and substitution operations are associative and commutative; so we can actually do them in any order. However, we need to know the segmentation first).

First we segment. There are eight possible segmentations, indicated by the ● below:

1. un Comité de sélection
2. un Comité de ● sélection
3. un Comité ● de sélection
4. un Comité ● de ● sélection
5. un ● Comité de sélection
6. un ● Comité de ● sélection
7. un ● Comité ● de sélection
8. un ● Comité ● de ● sélection

Only segmentations 6, 7, and 8 will lead to valid translations, since others contain phrases for which there is no entry in the phrase table. (This means that our probability model allocates some mass to segmentations that don't lead to a translation. But do not worry about this—this assumption just makes our model easier to work with).

Second, we permute the segments. The baseline decoder only considers the source language order, so we are left with just the segmentations 6, 7, and 8. We'll discuss swaps below, but I hope walking through a complete example of the baseline decoder first will be helpful.

Finally, we translate the segments independently. This leads to a large number of translations. For segmentation 6 alone, we get 8 translations:

1. a ● committee ● selection
2. a ● committee ● determining qualified
3. a ● Committee on ● selection
4. a ● Committee on ● determining qualified
5. an ● committee ● selection
6. an ● committee ● determining qualified
7. an ● Committee on ● selection
8. an ● Committee on ● determining qualified

We get four more for segmentation 7:

1. a ● committee ● for determining qualified
2. a ● Committee ● for determining qualified
3. an ● committee ● for determining qualified
4. an ● Committee ● for determining qualified

And 16 for segmentation 8:

1. a ● committee ● of ● selection
2. a ● committee ● of ● determining qualified
3. a ● committee ● to ● selection
4. a ● committee ● to ● determining qualified
5. a ● Committee ● of ● selection
6. a ● Committee ● of ● determining qualified
7. a ● Committee ● to ● selection
8. a ● Committee ● to ● determining qualified
9. an ● committee ● of ● selection
10. an ● committee ● of ● determining qualified
11. an ● committee ● to ● selection
12. an ● committee ● to ● determining qualified
13. an ● Committee ● of ● selection
14. an ● Committee ● of ● determining qualified
15. an ● Committee ● to ● selection
16. an ● Committee ● to ● determining qualified

For the case where adjacent phrases are swapped. Let's go back to step 2, and think about what this means for segmentation 6: un ● Comité de ● sélection. There are three possible orderings that swap adjacent phrases:

1. un ● Comité de ● sélection (the original order)
2. un ● sélection ● Comité de (swapping the 1st and 2nd phrase)
3. Comité de ● un ● sélection (swapping the 2nd and 3rd phrase)

In the full space of permutations, more orders are possible, but I'm not asking you to do this here (you might try it for the challenge part of the assignment). In particular, you don't need to consider any of these orders, which would involve swapping a phrase more than once:

- Comité de ● sélection ● un (to get this, you'd have to first swap un with Comité de, and then swap un again with sélection)
- sélection ● un ● Comité de (to get this, you'd have to first swap sélection with Comité de, and then swap sélection again with un)
- sélection ● Comité de ● un (similar to the above, also swapping un again with Comité de)

Now consider segmentation 7. Again, 3 permutations are possible:

1. un ● Comité ● de sélection (original order)
2. un ● de sélection ● Comité (swap 2nd and 3rd)
3. Comité ● un ● de sélection (swap 1st and 2nd)

And finally, for segmentation 8, 5 permutations are possible:

1. un ● Comité ● de ● sélection (original order)
2. un ● Comité ● sélection ● de (swap 3rd and 4th phrase)

3. un ● de ● Comité ● sélection (swap 2nd and 3rd phrase)
4. Comité ● un ● de ● sélection (swap 1st and 2nd phrase)
5. Comité ● un ● sélection ● de (swap 1st and 2nd phrase, and also swap 3rd and 4th phrase)

Now, for all permutations, you must consider all possible translations of the segments! This means that for segmentation 6, we get $3 \times 8 = 24$ possible translations, for segmentation 7 we get $3 \times 4 = 12$, and for segmentation 8, we get $5 \times 16 = 80$ possible translations. That's 116 possible translations! But you have probably noticed that many of these translations differ from others by only a single phrase or swap. Hence, dynamic programming will allow you to implement an efficient search by sharing work across translations.

Let's reason about what your the dynamic program should look like. Since only adjacent phrases can be swapped, and once swapped, a phrase cannot interact with any other, there are only two cases to reason about:

1. We can have a partial translation that completely covers a prefix of the source sentence. Denote by $h(i, e)$ the best partial translation of the first i French words ending in English word e. There are two ways to reach this situation:
    1. By extending a similar partial translation covering a shorter prefix; or
    2. By filling in a gap in the set of translated words that was created by translating the second in pair of phrases first. For example, in the third permutation of segmentation 8, there is a gap after we translate "de", because we have skipped translating "Comité". This gap is filled in when we translate "Comité".
2. We can have a partial translation that covers the first i words of the French, except those from k to j. This case is the antecedent of 1.2. above. This case can only result if we previously had a translation of the first $k - 1$ words, and then chose to translate a phrase from $j + 1$ to i, skipping the phrase from k to j. Let's denote the best possible translation of these words, ending in e, as $s(k, j, i, e)$.

If you're not sure where to start, you may want to read more about dynamic programming (https://people.eecs.berkeley.edu/~vazirani/algorithms/chap6.pdf).

Question 2 [15 marks]: Define a new dynamic program for the search space described above. More precisely: give correct recursive definitions for $h(i, e)$ and $s(k, j, i, e)$ described above.

You will need a precise mathematical description in order to convert it to code.

Question 3 [5 marks]: What is the computational complexity of your dynamic program? Justify your answer.

Question 4 [5 marks]: Define a mapping from hypothesis objects of your new dynamic program to stacks, so that all hypotheses in stack i represent translations of exactly i words. In other words: which stack should a hypothesis be placed on?

Question 5 [15 marks]: Implement your new dynamic program, by replacing the code from lines 42 through 76 of the default decoder.

HINT: This should require relatively little code; my implementation is about 15 lines longer than the default decoder. You will need to change the `state` object according to the new dynamic program that you've written, or you might use multiple types of `state` objects (but if you do, you'll need to check the type in `assign_stack` and `extend_state`). You will also need to think carefully about which stack to place new hypothesis objects in. If you've carefully answered the questions above, you should have a good idea about how to do this.

Question 6 [10 marks]: Using your new decoder, experiment with different combinations of values for the stack size parameter `-s` and the maximum number of translations `-k`, until you can no longer obtain any improvements.

- [4 marks] How do changes in these parameters affect the resulting log-probabilities?
- [2 marks] How do they affect speed?
- [2 marks] How do they affect the translations? Do the translations change? Do they make more sense? Are they more fluent? Give examples.
- [2 marks] What is the maximum log-probability you can obtain?

Part 3: Beyond local reordering [40 marks]

Implementing a decoder that can swap adjacent phrases and answering the accompanying questions will earn you 60 marks (i.e., a B). But swapping adjacent phrases will not get you anywhere close to the most probable translation according to this model. To get full credit, you **must** additionally design, implement, and experiment with a more advanced dynamic program. Specifically, I want you to design and implement a dynamic program that enforces a variant of what's called the *IBM constraint*. In this setting, the decoder may only translate a word at position i if all words to its left are already translated, except for at most one phrase. For example, suppose we the following sentence and segmentation:

il ● avait ● envoyé ● un ● remplaçant

Given this segmentation, we may choose to translate either "il" or "avait" first. If we translate "avait" first, we can then translate "envoyé", but we can't translate "un" or "remplaçant" yet, because in either case that would leave two phrases to the left untranslated. In fact, for this particular segmentation, there are sixteen possible orders, shown below with words that have been ordered before a word to their left in *italics*:

1. il ● avait ● envoyé ● un ● remplaçant
2. il ● avait ● envoyé ● *remplaçant* ● un
3. il ● avait ● *un* ● envoyé ● remplaçant
4. il ● avait ● *un* ● *remplaçant* ● envoyé
5. il ● *envoyé* ● avait ● un ● remplaçant
6. il ● *envoyé* ● avait ● *remplaçant* ● un
7. il ● *envoyé* ● *un* ● avait ● remplaçant
8. il ● *envoyé* ● *un* ● *remplaçant* ● avait
9. *avait* ● il ● envoyé ● un ● remplaçant
10. *avait* ● il ● envoyé ● *remplaçant* ● un
11. *avait* ● il ● *un* ● envoyé ● remplaçant
12. *avait* ● il ● *un* ● *remplaçant* ● envoyé
13. *avait* ● *envoyé* ● il ● un ● remplaçant
14. *avait* ● *envoyé* ● il ● *remplaçant* un ●
15. *avait* ● *envoyé* ● *un* ● il ● remplaçant
16. *avait* ● *envoyé* ● *un* ● *remplaçant* ● il

Before proceeding, you might find it helpful to convince yourself that other orders are not permitted under the above definition, and also that this search space is larger than the one in Part 2. Also notice where the above permutations are similar; this should help in designing your dynamic program.

Question 7 [10 marks]: Define a new dynamic program for the search space described above.

Question 8 [5 marks]: What is the computational complexity of your dynamic program? Justify your answer.

Question 9 [5 marks]: Define a mapping from hypothesis objects of your new dynamic program to stacks, so that all hypotheses in stack i represent translations of exactly i words. In other words: which stack should a hypothesis be placed on?

Question 10 [15 marks]: Implement your new dynamic program, by replacing the code from lines 42 through 76 of the default decoder.

Question 11 [5 marks]: Using your new decoder, experiment with different combinations of values for the stack size parameter `-s` and the maximum number of translations `-k`, until you can no longer obtain any improvements. Report the final result. What do you conclude?

Self-directed learning: beyond dynamic programming

In past versions of this course, this coursework contained an open-ended component; unfortunately, due to this year's enrollemnt it is not feasible for us to mark open-ended solutions, since it takes a lot of time. If you'd like to make your implementation closer to a real phrase-based decoder, you could add the ability to translate words within a particular distortion limit, as discussed in the video lecture. However, if you're keen to learn more about decoding, on your own you might consider implementing other approaches to solving the combinatorial optimization problem implied by the Viterbi approximation:

- Implement a greedy decoder (http://www.iro.umontreal.ca/~felipe/bib2webV0.81/cv/papers/paper-tmi-2007.pdf).
- Use chart parsing to search over many permutations in polynomial time (http://aclweb.org/anthology/C/C04/C04-1030.pdf).
- Use a traveling salesman problem (TSP) solver (http://aclweb.org/anthology-new/P/P09/P09-1038.pdf).
- Use finite-state algorithms (http://mi.eng.cam.ac.uk/~wjb31/ppubs/ttmjnle.pdf).
- Use Lagrangian relaxation (http://aclweb.org/anthology//D/D13/D13-1022.pdf).
- Use integer linear programming (http://aclweb.org/anthology-new/N/N09/N09-2002.pdf).
- Use A* search (http://aclweb.org/anthology-new/W/W01/W01-1408.pdf).

These methods all attempt to approximate or solve the Viterbi approximation to decoding. You can also try to approximate $p(\mathbf{e} \mid \mathbf{f})$ directly.

- Change the decision function (http://anthology.aclweb.org/N/N04/N04-1022.pdf) to minimize Bayes risk, which explicitly sums over translations.
- Use variational algorithms (http://aclweb.org/anthology//P/P09/P09-1067.pdf).
- Use Markov chain Monte Carlo algorithms (http://aclweb.org/anthology//W/W09/W09-1114.pdf).

But there are many other things you might try. There are many ways to decode, and good search is still considered an open problem.

Ground Rules

- You **must** work individually. If you submit work from someone other than yourself you will receive a mark of zero for the assignment. Your code and report must be your own work. You can safely assume that your instructor has software to accurately compute the probability that one piece of

code is a modified copy of the other. On the other hand, sharing questions, clarifications, and ideas that stop short of actual answers is fine and encouraged, especially through the forum (https://piazza.com/class/idfwi88bkpo377), since articulating your questions is often a good way to figure out what you do and don't know.

- You must submit five files. Any files with names other than these will be ignored.
    1. `answers.pdf` : A file containing your answers to Questions 1 through 11 in an A4 PDF. Your file must be written in LaTeX using this template (https://www.overleaf.com/latex/templates/infr1113-homework-1-template/bkqsppbgkqnn), which you should clone and edit to provide your answers. Answers provided in any other format will receive a mark of zero. Your answers **must not exceed two pages**, so be concise. You are however permitted to include graphs or tables on an unlimited number of additional pages. They should be readable. They should also be numbered and the text should refer to these numbers.
    2. `part2.out` : The highest-probability translations of the input that you can produce using your Part 2 decoder.
    3. `part2.py` : Your implementation of the Part 2 decoder.
    4. `part3.out` : The highest-probability translations of the input that you can produce using your Part 3 decoder.
    5. `part3.py` : Your implementation of the Part 3 decoder.
- Questions that are not answered will receive no marks. Even if you aren't sure of your answer, an attempt may earn partial marks if it shows evidence of understanding.
- You are strongly encouraged to comment your code; markers may review comments if they can't understand what your code does, so this is in your best interests.
- Your name **must not appear in any of the submitted files**. If your name appears in the code or pdf (or output) you will receive a mark of zero.

To submit your files on dice, run:

```
submit mt 1 answers.pdf part2.py part3.py part2.out part3.out
```

Acknowledgements

This assignment was developed and refined over several years in collaboration with Chris Callison-Burch (http://www.cis.upenn.edu/~ccb/), Chris Dyer (http://www.cs.cmu.edu/~cdyer), and Matt Post (http://cs.jhu.edu/~post/).

Footnotes

1. the machine translation jargon for a sequence of words is *phrase*, but these aren't phrases in any linguistic theory—they can be any sequence of words in the sentence. ↩
2. Technically, this makes the default decoder a *hidden semi-Markov model* ↩
3. Sometimes these problem combine to create a *fortuitous search error* (https://www.aclweb.org/anthology/P/P01/P01-1030.pdf), where inexact search finds a better translation than the one preferred by a poor model. We will not try to cause fortuitous search errors! ↩

Informatics Forum, 10 Crichton Street, Edinburgh, EH8 9AB, Scotland, UK
Tel: +44 131 651 5661, Fax: +44 131 651 1426, E-mail: school-office@inf.ed.ac.uk