LSA 354: Statistical Parsing: Problem Set

[Some of these problems are borrowed with thanks from Michael Collins and Regina Barzilay.]

Question 1

Let's use a very simple grammar (where S is the start symbol, and terminals are shown in italic)

S	$\longrightarrow$	NP VP	N	$\longrightarrow$	*cats*
VP	$\longrightarrow$	V	N	$\longrightarrow$	*claws*
VP	$\longrightarrow$	V NP	N	$\longrightarrow$	*people*
NP	$\longrightarrow$	NP RelCl	Comp	$\longrightarrow$	*that*
NP	$\longrightarrow$	N	V	$\longrightarrow$	*scratch*
RelCl	$\longrightarrow$	Comp V	V	$\longrightarrow$	*bite*

Simple (non-tabular) parsing strategies, or in particular, recognizers,[1] can be implemented using a state representation where there is a list (stack or queue) for elements being processed and another for the input sentence. We will assume that sort of representation for the questions here, so a state can be represented on one line. Since drawing search trees can get rather difficult with such representation, we can use two devices:

- We can draw the trees as an indented list, where the indentation of the lines shows the depth of the tree. (This is an especially natural way to think of things when used with depth-first search, since then the order of lines down the page corresponds to the order in which things are done depth first, but it can equally represent any other navigation of a search tree: it's just a tree on its side.)
- We can assume an 'oracle' which at any point chooses only the move (or moves) that leads to a successful parse of the sentence. This allows us to abbreviate things by leaving out all the dead ends. If a sentence only has one parse, or we are only showing one parse, we can also omit the indentation. This corresponds to showing the succession of states down one branch of the search tree - we're ignoring search control, and just showing the moves that we're interested in.

Important notation: We use uppercase latin for nonterminals, lowercase latin for terminals, and greek letters for possibly empty sequences of categories (terminal or nonterminal) - except for α which is either a single category or the empty string, and *X* which can be either a nonterminal or a terminal. We use a bar over categories or sequences of categories that are predicted versus categories that have been found.

Here is a method for LR (bottom-up shift-reduce) parsing, where we want to parse a string given in the *input* as a particular category. Note that it assumes that the right hand end of a list of symbols counts as the top of the stack. One starts with the initialization in Begin, and then one can do any of the other moves any number of times in any order. Parsing is successful if the sequence of moves ends with a state that fulfills the Halt criterion. Combining these moves with a search algorithm would give an LR parser.

[1] That is, devices that can simply determine whether a list of words is in the language defined by the grammar, but don't provide parse trees.

Begin Place the predicted start symbol on top of the stack (i.e. with a bar over it)

- (Shift) Put the next input symbol on top of the stack
- (Reduce) If γ is on top of the stack and $A \longrightarrow \gamma$, replace γ with A
- (Reduce attach) If $\overline{A}\gamma$ is on top of the stack and $A \longrightarrow \gamma$, remove $\overline{A}\gamma$.

Halt Halt with a success if the stack is empty and there is no more input

For instance, here is a successful parse of *cats scratch people*

Stack	Input	Operation
$\overline{S}$	*cats scratch people*	Begin
$\overline{S}$ *cats*	*scratch people*	Shift
$\overline{S}$ N	*scratch people*	Reduce
$\overline{S}$ NP	*scratch people*	Reduce
$\overline{S}$ NP *scratch*	*people*	Shift
$\overline{S}$ NP V	*people*	Reduce
$\overline{S}$ NP V *people*		Shift
$\overline{S}$ NP V N		Reduce
$\overline{S}$ NP V NP		Reduce
$\overline{S}$ NP VP		Reduce
		Reduce attach
		Halt

1. (Warm-up question!) Suppose we didn't have a start symbol, we just wanted to know if there is *some* category from which the whole string can be generated. This makes some linguistic sense. Sometimes a 'sentence' in a newspaper (or the Stanford Bulletin) is just a noun phrase, and people commonly respond to questions (like *When will you get around to the assignment?*) with fragments such as PPs (*During next week.*). How would one modify [hint: simplify!] the above parser specification for this case?

2. Top-down parsing

   (a) Give a similar specific set of operations that implement a top-down, left-to-right (LL) parser (Hint: you will want to predict a lot more categories than in the above example!)

   (b) Trace the successful parse move sequence for the sentence *cats scratch people* using the grammar above. Assume that we are literally doing top-down parsing right to the level of guessing words.

   (c) Suppose we added one or more rules that rewrote things as empty to the grammar. (In particular, you might consider the rule NP $\longrightarrow$ e, and parsing the sentence *cats bite*.) Does your top-down parser need moficiation to handle this case correctly? If so, how? If not, say briefly why not?

3. We noted that top-down parsers have a problem with left-recursion rules. For example, they will have problems with the NP $\longrightarrow$ NP RelCl rule in the grammar above. Joe Genius notes that this problem has a ready solution for this grammar. He rejigs his top-down parser to expand rules from right-to-left, rather than left-to-right. Think about how this would fix the problem.

(a) Since Joe is a genius, he knows that this isn't a full solution to the problem, because his new parser will have problems with right recursive rules like S $\longrightarrow$ AdvP S. (For instance, we might suggest such a rule to parse [*Most of the time] I go to class*.) However, he comes up with the following ingenious idea. For each grammar rule, if it is left recursive, he will expand its categories from right-to-left, otherwise, he will expand it from left-to-right, as previously. (Carrying this idea through necessitates a number of further complexities involving maintaining a stack of parts of the sentence that one has parsed, but we'll leave the details to Joe to work out.) Is this a full solution to the problem of edge-recursion? If not, what kinds of grammar rules will still cause problems for Joe's parser? Give a realistic example of a place in English grammar where this problem turns up.

(b) We presented the problem with top-down parsing in terms of a problem handling left-recursion rules of the form $X \longrightarrow X\ \gamma$. But the problem is actually a bit more general than that. What is the more general statement of when a top-down parser runs into trouble?

4. Here are the corresponding moves for a left-corner parser. This time we have to regard the left end of any lists as the top of the stack. (It's common to need to change this convention, so that we can use grammar rules to match sequences in different orders without having to reverse them, which makes them hard to read.)

Begin Place the predicted start symbol on the stack.

- (Shift) Put the next input symbol on top of the stack.
- (Pop) If $\overline{\alpha}$ is on top of the stack, and α is the next input symbol, remove both.
- (Leftcorner attach) If $X\overline{A}$ is on top of the stack and $A \longrightarrow X\gamma$, replace $X\overline{A}$ by $\overline{\gamma}$.
- (Leftcorner) If X is on top of the stack and $A \longrightarrow X\gamma$, replace X by $\overline{\gamma}A$.

Halt Halt with success if the stack is empty and there is no more input.

The idea of a left-corner parser is it mixes top-down and bottom-up processing: it both works predictively down from a goal, and up from an identified left-corner of a phrase.

(a) Trace the moves of one successful parse of the sentence *cats scratch people that bite* using the grammar at the beginning.

(b) Through the "Leftcorner attach" operation, this parser does what is sometimes referred to as "composition." Namely, if it has found a Det, and the goal is an $\overline{NP}$, and there is a rule NP $\longrightarrow$ Det Adj N, then it will in one step remove the Det and the $\overline{NP}$ goal, and put goals of an $\overline{Adj}$ and $\overline{N}$ on the stack. A completed NP never actually appears on the stack. Change the above parser so that it doesn't do composition. That is, the "Leftmost attach" operation would be removed, and the completed NP would be placed on the stack using the exisiting Leftcorner operation. Modify the parser as needed so as to still have a sound and complete parser.

5. All of the methods we have looked at above can be regarded as instances of "Generalized Left Corner Parsing" (A.J. Demers, Generalized left corner parsing, *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pp. 170-181, 1977). Here's the algorithm:

Begin Place the predicted start symbol on the stack (i.e., with a bar over it.)

- (Shift) Put the next input symbol on top of the stack

- (LeftPart) If β is on top of the stack and condition ϕ holds and $A \longrightarrow \beta\gamma$, replace β by $\overline{\gamma}A$
- (Attach) If $X\overline{X}$ is on top of the stack, remove both.

Halt Halt with success if the stack is empty and there is no more input

All of the top-down (LL), bottom-up (LR), and left-corner (LC) parsing can be realized by the above algorithm, by appropriately defining the condition ϕ (which might depend on β, γ, etc.). Define what ϕ should be to give each of these strategies.

Question 2

A probabilistic context-free grammar $G = (N, \Sigma, R, S, P)$ in Chomsky Normal Form is defined as follows:

- N is a set of non-terminal symbols (e.g., `NP, VP, S` etc.)
- Σ is a set of terminal symbols (e.g., *cat, dog, the*, etc.)
- R is a set of rules which take one of two forms:
  - $X \rightarrow Y_1Y_2$ for $X \in N$, and $Y_1, Y_2 \in N$
  - $X \rightarrow Y$ for $X \in N$, and $Y \in \Sigma$
- $S \in N$ is a distinguished start symbol
- P is a function that maps every rule in R to a probability, which satisfies the following conditions:
  - $\forall r \in R,\ P(r) \geq 0$
  - $\forall X \in N,\ \sum_{X \rightarrow \alpha \in R} P(X \rightarrow \alpha) = 1$

Now assume we have a probabilistic CFG G', which has a set of rules R which take one of the two following forms:

- $X \rightarrow Y_1Y_2 \ldots Y_n$ for $X \in N$, $n \geq 2$, and $\forall i,\ Y_i \in N$
- $X \rightarrow Y$ for $X \in N$, and $Y \in \Sigma$

Note that this is a more permissive definition than Chomsky normal form, as some rules in the grammar may have more than 2 non-terminals on the right-hand side. An example of a grammar that satisfies this more permissive definition is as follows:

S	$\rightarrow$	NP	VP		0.7
S	$\rightarrow$	NP	NP	VP	0.3
VP	$\rightarrow$	Vt	NP		0.8
VP	$\rightarrow$	Vt	NP	PP	0.2
NP	$\rightarrow$	DT	NN	NN	0.3
NP	$\rightarrow$	NP	PP		0.7
PP	$\rightarrow$	P	NP		1.0

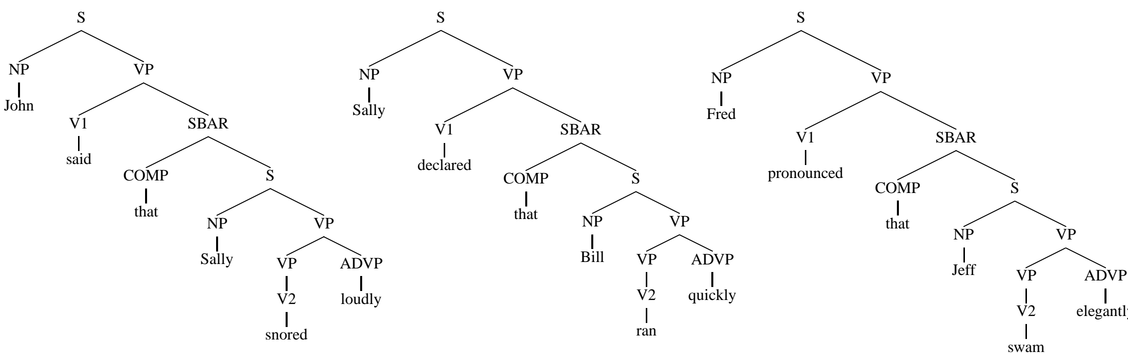
Vt	$\rightarrow$	saw	1.0
NN	$\rightarrow$	man	0.7
NN	$\rightarrow$	woman	0.2
NN	$\rightarrow$	telescope	0.1
DT	$\rightarrow$	the	1.0
IN	$\rightarrow$	with	0.5
IN	$\rightarrow$	in	0.5

Question 2(a): Describe how to transform a PCFG G', in this more permissive form, into an "equivalent" PCFG G in Chomsky normal form. By equivalent, we mean that there is a one-to-one function f between derivations in G' and derivations in G, such that for any derivation T' under G' which has probability p, $f(T')$ also has probability p. (Note: one major motivation for this transformation is that we can then apply the dynamic programming parsing algorithm, described in lecture, to the transformed grammar.) Hint: think about adding new rules with new non-terminals to the grammar.

Question 2(b): Show the resulting grammar G after applying your transformation to the example PCFG shown above.

Question 3

Nathan L. Pedant decides to build a treebank. He finally produces a corpus which contains the following three parse trees:



Clarissa Lexica then purchases the treebank, and decides to build a PCFG, and a parser, using Nathan's data.

Question 3(a): Show the PCFG that Clarissa would derive from this treebank.

Question 3(b): Show two parse trees for the string "Jeff pronounced that Fred snored loudly", and calculate their probabilities under the PCFG.

Question 3(c): Clarissa is shocked and dismayed, (see 2(b)), that "Jeff pronounced that Fred snored loudly" has two possible parses, and that one of them—that Jeff is doing the pronouncing loudly—has relatively high probability, in spite of it having the `ADVP` *loudly* modifiying the "higher" verb, *pronounced*. This type of high attachment is never seen in the corpus, so the PCFG is clearly missing something. Clarissa decides to fix the treebank, by altering some non-terminal labels in the corpus. Show one such transformation that results in a PCFG that gives zero probability to parse trees with "high" attachments. (Your solution should systematically refine some non-terminals in the treebank, in a way that slightly increases the number of non-terminals in the grammar, but allows the grammar to capture the distinction between high and low attachment to VPs.)

Question 4

Recall the definition of a PCFG in Chomsky normal form from question 1. Now assume we have a probabilistic CFG G', which has a set of rules R which take one of the three following forms:

1. $X \rightarrow Y_1 Y_2$ for $X \in N$, and $Y_1, Y_2 \in N$

2. $X \rightarrow Y$ for $X \in N$, and $Y \in \Sigma$

3. $X \rightarrow Y$ for $X \in N$, and $Y \in N$

Note that this is very similar to a Chomsky normal form grammar, but that we are now allowed rules of form (3), such as `S -> VP`, where there is a single symbol on the right-hand-side of the rule, and this symbol is a non-terminal. We will refer to these new rules as **unary productions**. (Note that productions of the form in (2), such as `N -> dog`, will *not* be referred to as unary productions, as their right-hand-side is a terminal symbol.) We will refer to rules captured by cases (1) and (2) as **non-unary productions**.

As one example, the following grammar contains unary productions:

S	→	NP	VP	0.7
S	→	SBAR	VP	0.3
VP	→	Vi		0.4
VP	→	Vt	NP	0.4
VP	→	V3	SBAR	0.2
NP	→	NN		0.3
NP	→	DT	NN	0.7
SBAR	→	COMP	S	0.6
SBAR	→	S		0.4

Vi	→	sleeps	1.0
Vt	→	saw	1.0
V3	→	said	1.0
NN	→	man	0.7
NN	→	woman	0.2
NN	→	telescope	0.1
DT	→	the	1.0
COMP	→	that	1.0

In this question, we'll attempt to convert a PCFG with unary productions into an "equivalent" PCFG which is in Chomsky normal form (note that we'll have to be careful with what we mean by "equivalent", we'll come to this shortly).

As a first step, we will use the classic transformation for (non-probabilistic) context-free grammars, that results in a new grammar that accepts the same set of strings as the original grammar, but which has all unary productions removed. Applying this transformation to the grammar above results in the CFG shown in figure 1 (note that the probabilities are missing – we'll fill them in soon).

S	→	NP	VP
S	→	SBAR	VP
VP	→	sleeps	
VP	→	Vt	NP
VP	→	V3	SBAR
NP	→	man	
NP	→	woman	
NP	→	telescope	
NP	→	DT	NN
SBAR	→	COMP	S
SBAR	→	NP	VP
SBAR	→	SBAR	VP

Vi	→	sleeps
Vt	→	saw
V3	→	said
NN	→	man
NN	→	woman
NN	→	telescope
DT	→	the
COMP	→	that

Figure 1: A transformed grammar, G'

This grammar transformation works in the following way. We form a new grammar G' from an existing grammar G by first taking all non-unary rules from G. Then, if there is any sequence of n productions in G

$$B_0 \rightarrow B_1 \rightarrow B_2 \ldots B_{n-1} \rightarrow \alpha$$

such that $B_i \rightarrow B_{i+1}$ for $i = 0 \ldots n-2$ are unary productions in the grammar, and $B_{n-1} \rightarrow \alpha$ is a non-unary production, then we add the rule

$$B_0 \rightarrow \alpha$$

to G'. For example, in the above example we have the sequence

$$\text{SBAR} \rightarrow \text{S} \rightarrow \text{NP VP}$$

so we add the rule SBAR $\rightarrow$ NP VP to G'. As another example, we have the sequence

$$\text{VP} \rightarrow \text{Vi} \rightarrow \text{sleeps}$$

so we add the rule VP $\rightarrow$ sleeps to G'.

We now come to the definition of equivalence. We will say that the PCFG G' is "equivalent" to a PCFG G if:

- For any string w, if $T(w)$ is the highest probability parse tree for w under the grammar G, and $T'(w)$ is the highest prob. parse under G', then these two parse trees have the same probability under their respective grammars.
- There is a function f such that $T(w) = f(T'(w))$. i.e., there is a function such that the highest probability parse tree in the original grammar can be recovered from the highest probability parse tree under G'.
- In some cases, we will allow the PCFG G' to be **deficient**. This means that we will relax the requirement on probabilities on rules to satisfy the condition $\forall X \in N, \sum_{X \rightarrow \alpha \in R} P(X \rightarrow \alpha) < 1$ rather than $\forall X \in N, \sum_{X \rightarrow \alpha \in R} P(X \rightarrow \alpha) = 1$, as in the definition in question 1.

Question 4(a): Add probabilities to the grammar in figure 1, so that the new PCFG G' is equivalent to the old PCFG G. Describe the function f that maps a parse tree in G' to a parse tree in G.

Question 4(b): Describe a strategy for creating an equivalent PCFG G' in Chomsky normal form for *any* PCFG G in the form described at the start of this question (i.e., a PCFG that may have unary productions in addition to Chomsky normal form rules). You may assume that for any unary rule, its probability is strictly less than 1. Describe also the function f used to recover the highest probability tree under G from the highest probability tree under G'. *Note that your resulting PCFG G' may be deficient in some cases.* Illustrate your transformation on the two PCFGs shown in figure 2 (these grammars will help you, in terms of illustrating some "tricky" cases that you'll run into with unary productions). **Hint: remember throughout this question that the goal of G' is to allow recovery of the maximum probability parse under G.**

Question 5: CKY Parsing of PCFGs

One of the most famous ambiguous sentences of English from early computational linguistics work is:

Grammar 1:

S	→	NP	VP	0.7
S	→	SBAR	VP	0.3
VP	→	Vi		0.4
VP	→	Vt	NP	0.4
VP	→	V3	SBAR	0.2
NP	→	NN		0.3
NP	→	DT	NN	0.7
SBAR	→	COMP	S	0.4
SBAR	→	S		0.5
SBAR	→	NP	VP	0.1

Vi	→	sleeps	1.0
Vt	→	saw	1.0
V3	→	said	1.0
NN	→	man	0.7
NN	→	woman	0.2
NN	→	telescope	0.1
DT	→	the	1.0
COMP	→	that	1.0

Grammar 2:

S	→	NP	VP	0.7
S	→	SBAR	VP	0.2
S	→	SBAR		0.1
VP	→	Vi		0.4
VP	→	Vt	NP	0.4
VP	→	V3	SBAR	0.2
NP	→	NN		0.3
NP	→	DT	NN	0.7
SBAR	→	COMP	S	0.4
SBAR	→	S		0.5
SBAR	→	X		0.1
X	→	S		0.1
X	→	NP	NP	0.9

Vi	→	sleeps	1.0
Vt	→	saw	1.0
V3	→	said	1.0
NN	→	man	0.7
NN	→	woman	0.2
NN	→	telescope	0.1
DT	→	the	1.0
COMP	→	that	1.0

Figure 2: Two grammars with unary productions

Time flies like an arrow.

Using the ambiguity-causing-things that we have standardly used (part of speech, PP attachment, noun compounding), you should be able to generate *four* parse trees (and, hence, different meanings) for this sentence. If you're having trouble thinking of four, or just if you're interested in the history of this sentence, you can look here:

http://mitpress.mit.edu/e-books/Hal/chap7/seven2.html

Assume the following grammar G:

S	$\rightarrow$	NP VP	0.8	N	$\rightarrow$	*time*	0.5
S	$\rightarrow$	VP	0.2	N	$\rightarrow$	*flies*	0.3
VP	$\rightarrow$	V NP	0.5	N	$\rightarrow$	*arrow*	0.2
VP	$\rightarrow$	V PP	0.3	V	$\rightarrow$	*time*	0.3
VP	$\rightarrow$	VP PP	0.2	V	$\rightarrow$	*flies*	0.3
NP	$\rightarrow$	Det N	0.3	V	$\rightarrow$	*like*	0.4
NP	$\rightarrow$	N	0.3	P	$\rightarrow$	*like*	1.0
NP	$\rightarrow$	N N	0.2	Det	$\rightarrow$	*an*	1.0
NP	$\rightarrow$	NP PP	0.2				
PP	$\rightarrow$	P NP	1.0				

1. Draw the four parses for the sentence using the original grammar above.
2. Generate a version of this grammar in Chomsky Normal Form by performing unary rule removal. Make the probabilities of the rules such that the language generated by the grammar is unchanged (sentences get the same probability). (I.e., work out what to do with rule probabilities when unary removal is done.)
3. Draw a CKY parse triangle for this grammar parsing the sentence above, including working out the inside probabilities. What is $P(\textit{Time flies like an arrow}|G)$?
4. Draw a CKY parse triangle for this grammar parsing the sentence above, working out Viterbi (maximum) probabilities. What is $\arg\max_t P(t|\textit{Time flies like an arrow}, G)$? Indicate which of the four parses in (a) is the one chosen as most likely.
5. What would be the effect on the probabilities calculated in the previous part of adding some other preposition, say P $\rightarrow$ *on*, with some reasonable non-zero probability? Discuss in a sentence or two whether this seems linguistically sensible in terms of using PCFGs for disambiguating sentences?

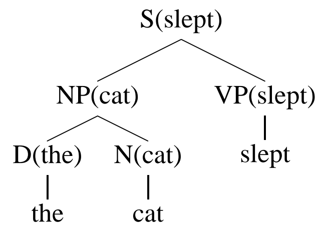
Question 6

We will refer to a "lexicalized PCFG" in Chomsky normal form, as a PCFG $G = (N, \Sigma, R, S, P)$ similar to that in question 1, where each of the rules in R takes one of the following three forms:

- $X(h) \rightarrow Y_1(h)\ Y_2(w)$ for $X \in N$, and $Y_1, Y_2 \in N$, and $h, w \in \Sigma$.
  e.g., `NP(man)` $\rightarrow$ `NP(man) PP(with)`.
- $X(h) \rightarrow Y_1(w)\ Y_2(h)$ for $X \in N$, and $Y_1, Y_2 \in N$, and $h, w \in \Sigma$.
  e.g., `S(snores)` $\rightarrow$ `NP(man) VP(snores)`.
- $X(h) \rightarrow h$ for $X \in N$, and $h \in \Sigma$
  e.g., `NP(man)` $\rightarrow$ `man`.

Here the symbols in the grammar rules are of the form $X(h)$, where X is a symbol such as `NP`, `VP`, etc., and h is a lexical item such as *man*, *snores*, etc.

In addition, for any symbol of the form $X(h)$, there is a probability $P_S(X(h))$ which is the probability of $X(h)$ being chosen as the root of a parse tree. As one example, the tree



would have probability

$P_S(\texttt{S(slept)})\times$
$P(\texttt{S(slept)} \rightarrow \texttt{NP(cat) VP(slept)}|\texttt{S(slept)})\times$
$P(\texttt{NP(cat)} \rightarrow \texttt{D(the) N(cat)}|\texttt{NP(cat)})\times$
$P(\texttt{D(the)} \rightarrow \texttt{the}|\texttt{D(the)})\times$
$P(\texttt{N(cat)} \rightarrow \texttt{cat}|\texttt{N(cat)})\times$
$P(\texttt{VP(slept)} \rightarrow \texttt{slept}|\texttt{VP(slept)})$

Question 6(a): Describe a dynamic programming algorithm, similar to the one in lecture, which finds the highest scoring parse tree under a grammar of this form. Your algorithm should make use of a dynamic programming table $\pi[i, j, k, X]$ where

$\pi[i, j, k, X]$ = highest probability for any parse tree whose root is the symbol $X(w_k)$, and which spans words $i \dots j$ inclusive

For example, if the sentence being parsed is $w_1, w_2, \dots w_6$ = *the cat sat on the mat*, then $\pi[4, 6, 4, \texttt{PP}]$ would store the maximum probability for any parse tree whose root is PP(on), and which spans the string *on the mat*.

Note: your algorithm should also allow recovery of the parse tree which achieves the maximum probability.

Question 6(b): What is the running time of your algorithm?