Short Version

Read in a bunch of sentences, prefixed by a weight and separated by newlines, followed by a newline, followed by a phrase. List all possible completions of that phrase sorted by weight. Use a clever type of tree to make this somewhat efficient. Print out the internal node of highest degree and its number of children to prove you used the right data structure.

Tries and Autocomplete

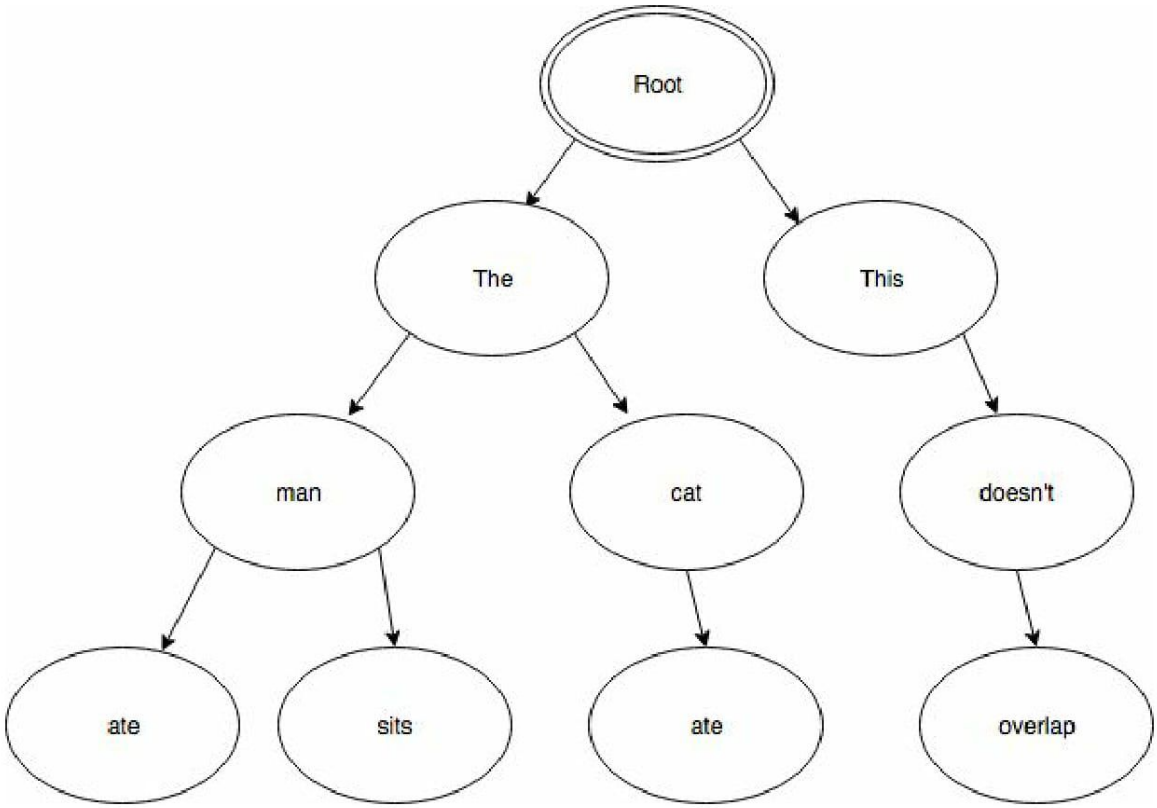
A Trie is a special type of tree that acts as a type of associative array (like a HashMap in Java, or dict in python). But you wouldn't likely use a Trie if you just wanted t.put(key, value) and value = t.get(key). You already know that a hash table provides that functionality rather efficiently. However, what if the key was a string, and you wanted to autocomplete the string interactively? For example, a toy store might have a search field, and when someone types "Thomas the ", you can look through your Trie of keys and see that there are keys that begin with that and end with "Tank Engine toys" and "Tank Engine videos".
For more on Tries, start with this Wikipedia entry:
https://en.wikipedia.org/wiki/Trie

Diversion

There are many variations of Tries that provide richer functionality. The first you might consider is a suffix tree, which stores all suffixes of a string. Assuming we are still interested in sentences broken up by words (not down to individual letters), the suffixes for "The man ate" would include "The man ate", "man ate", and "ate". For storing sets of strings instead of a single string, you would use a generalized suffix tree. A generalized suffix tree is just a suffix tree that uses an additional field to track which sentence ended which suffix.
A generalized suffix tree efficiently allows autocomplete even if you missed the start of the sentence. Furthermore, the overlaps of partial paths in the tree would represent common phrases (not just suffixes) shared by multiple sentences. You could efficiently find the most often repeated phrase that was 4 words (or longer). You could efficiently find the longest shared phrase present in 2 (or more) sentences. Again, more information is available at Wikipedia:
https://en.wikipedia.org/wiki/Suffix_tree
https://en.wikipedia.org/wiki/Generalized_suffix_tree
You will not need these variants in this assignment. You will, however, in assignment 2.

Longer Version

You will use a Trie to store a set of sentences and a score/weight for each sentence. Starting with the root node, each child is a word that represents the starting word for each sentence. A node b will have children for each sentence that shares a prefix from the root to b.



The tree above is a Trie (without weights) for the sentences:

- The man ate
- The man sits
- The cat ate
- This doesn't overlap

Note that despite the fact that "The man ate" and "The cat ate" share the word "ate", they do not merge their paths. The only shared part is the single-word prefix: "The". Merging after splitting would make storing a value at the leaf node more complicated (also, the result would be a graph, but not a tree).

Example input/output

If the input looked like (note the empty line):

```
2 The man ate
5 The man SITS
1 The cat ate
777 This doesn’t overlap

The
```

Your code should return

```
man SITS
man ate
cat ate
man 2
```

This is every sentence completion for the provided prefix, sorted by the weight. Since the standard C++ tolower is an awful function, you can treat words with different capitalization as different words.
As you build the Trie, keep track of when you add children to a node and how many children it has. The last line is “man 2”, which would have been the max degree node as you were building the tree (note that “man” became the max node before “The” tied it, so please do your comparison for max with current > max, not current >= max).
You are restricted to the standard library. Your Trie implementation must be your own.
All input will be via standard in. Output will be to standard out.
Special case: if there are no completions, output “No valid completion” and eol.
The input grammar is roughly:

Input = WeightedSentences EmptyLine Prefix NEWLINE
WeightedSentences = (INT SPACE Sentence NEWLINE)+
Sentence = (WORD SPACE)* WORD
EmptyLine = NEWLINE NEWLINE
Prefix = (WORD SPACE)* WORD

Yes, the grammar for Prefix is the same as Sentence.
The tokens:

INT = valid C++ integer recognized by std::cin >> number;
NEWLINE = good ol’ \n
SPACE = ASCII 32 (normal ‘ ‘ space)
WORD = valid C++ string that contains no newlines nor spaces

I have made these decisions based on what was easiest using just the standard library and the default behavior of cin and cout.
The output grammar should be:
Output = ((Sentence NEWLINE)+ | ComplaintOutput) MaxNode
MaxNode = WORD INT NEWLINE
ComplaintOutput = "No valid completion" NEWLINE

Grading

Your code will be run against a number of examples. The first test will verify that your prefix matching is correct. The second will test an unmatchable prefix. A third set of tests will verify the sort order of the output. If you just output the complaint string for all inputs, you will not get credit for that.
We will use some rather long input lines.
Reminder: if your code does not compile and run, you will receive no credit. Code will be compiled by typing "make" in the directory of your project. Please have the output executable named prog1.
Input will be via standard in. If you want to test, try (make testfile.txt contain the input example above).

```
cat testfile.txt | prog1 > output.txt
```