

*Mini-Project: Hot Phrase Scores***Short task statement:**

Do the task of Assignment 4, but searching for whole phrases with punctuation stripped off as in Assignment 5. Instead of just counting occurrences, compute two kinds of “hot scores”: the number of words matched, and the total number of characters in those words. We will use Rolling Stone’s list of 500 Top Songs for the hot-phrases, and will pull various texts off the Web. For now, “occurrence” will mean equality—see questions below for a preview of the idea (to be implemented later) of matching some of a phrase/song-title and/or some of an individual word. Finally sort the phrases (song titles) found to occur and print them with the highest number-of-words hot score first, and then an overall hot score defined to be the sum of those scores divided by the number of words in the text file, multiplying by 100.0 to make it look like a percentage. Do the same for the number-of-chars hot-score, dividing by the number of chars in the text file.

Other Rules:

Your code for m hot-phrases and n text-words must run in time $o(mn)$ as demonstrated analytically by comments in your code (drawing on your Assignment 5 essay questions). You may assume that calls to the C++ standard library `sort` function, which requires including `<algorithm>`, on N items run in $O(N \log N)$ time. Modify your `Phrase` class as-needed and submit four files (or six with `StringWrap` too) as on Assignment 5, but with `HotPhrases.cpp` and `HotPhrases.make` for the files, and `hotphrases` for the named executable produced by the makefile (no `.exe` ending in UNIX). Now there must be an `operator<` function inside the `PhraseNNN.{h,cpp}` files but outside the `Phrase` class—it should do the same thing as your prior `lessThan`, and depending on how you code things, may-or-may-not need to be declared a `friend` by the class.

There is one particular “modularity” rule for the `Phrase` class. It must *not* maintain information about a particular hot-score, because that pertains to a particular text file that is not logically part of what the class is modeling. You are, however, welcome to derive a class `HotPhrase` from it to include this extra information. Alternatively, you can make `HotPhrase` a “wrapper” class for a phrase that adds this extra functionality—this is like what `StringWrap` does for `string`. The buzzword for the latter is “using composition not inheritance,” but it may involve more work because you may have to create some “delegating” versions of methods of the `Phrase` class from scratch, whereas with the derived-class approach you can just inherit them. Either way you must create the `HotPhrase` class (it can go in the `PhraseNNN.{h,cpp}` files) and describe in a report question how you handled it.

For the C++ library `sort` function there are issues that may differ between `timberlake` and your home compiler(s). These are illustrated by lines commented out in my answer-key code, where I’ve read the phrase file into a `vector<Phrase>` called `phrases`. First, below my `Phrase` class I have:

```
bool operator<(const Phrase& lhs, const Phrase& rhs); //== what you should do
bool lt(const Phrase& lhs, const Phrase& rhs);        //to fix syntactic hitch
```

Then in my client driver I have:

```
//sort(phrases.begin(), phrases.end(), Phrase::lessThan); //error
//Not allowed to pass a non-static member function.
//sort(phrases.begin(), phrases.end(), operator<); //error :-(
//sort(phrases.begin(), phrases.end(), less<Phrase>); //error :-( :-(
//sort(phrases.begin(), phrases.end(), lt); //OK, non-operator lt
sort(phrases.begin(), phrases.end()); //OK!---which IMHO is wrong
```

Similar considerations apply to your sorts on the hot-score criteria. Try your code with `RS500songs.txt` as phrase file and various other texts in the `Java2C++` directory, e.g. find the longest Beatles song in `Hamlet.txt` or see if `Gettysburg.txt` is hip at all. 90 pts. for code, plus 30 for having *a few good* INV/REQ/ENS comments and for these two report questions, for 120 points total:

1. How did you handle the `HotPhrase` class? Did you use inheritance (i.e., subclassing/deriving) or composition (aka. “wrapping”)? What did you do to implement the sorts, and how did you make sure they displayed in highest-to-lowest order? (12 pts.)
2. How might you modify your code to detect, for example, text words in `my lifetime` as a match to the song `In My Life`, or `I walked the line` as a Johnny Cash reference despite the extra `ed`? In particular, how might you modify the `prefixOf` method? Mind you, `lit my fire` might not be caught as a match for `Light My Fire`, and there are other problems—but doing at least this much would seem worth it, so it’s worth a few sentences of thinking about it. (9 pts.) (which means 9 pts. for the comments)

Supplementary Directions

Your `Phrase` class *must* include a method `bool prefixOf(const Phrase& rhs)` that returns `true` if and only if this-phrase is a prefix of `rhs`. (This is as a result of this method being taken off the exam. Even if you differ from the following you will want this method.)

It is *recommended* that you adopt the following strategy, which is basically answer (A) on the posted Assgt. 5 key taken from essay problem (1) into the situation of essay (2):

1. Compute p = the maximum number of words in a phrase that was read, while or after reading all the phrases into a `vector<Phrase>`.
2. Read the text words into a `vector<string>`, and then make a `vector<Phrase>` by taking every p consecutive words in the text, making a `Phrase` from them, and using `push_back` to append it to the vector. Finally append the phrase made from the last $p - 1$ words, then the last $p - 2$, down to a final one-word phrase from the last word.
3. Sort both vectors of phrases according to your `operator<` as indicated above. (Your home-machine compiler may differ here!?)
4. For every phrase `s` in the first vector, move forward in the text-phrase vector only as far as that phrase could possibly go in alphabetical order as defined by your `lessThan`. For every text phrase `t` it could possibly match, call `s.prefixOf(t)`. If that is true, it means the phrase `s` occurs at that point in the text (ignoring possible extra words if `s` is shorter than p words). Then compute the appropriate hot-score(s), make `HotPhrase` object(s) out of the result, and add them to a `vector<HotPhrase>`.
5. Finally sort the `HotPhrase` `vector`(s) according to the two criteria—for which it appropriate to use alphanumeric names rather than an operator name which would clash with each other!—and print the results and overall score on the text file.

It is true that binary search avoids the $n \log n$ for sorting the text-phrase vector, which is the slowest term here—and technically $n \log n$ requires n to be less than exponential in m to be $o(nm)$ though that is a fine assumption. However, this is a situation of using binary search to look for a *possible place* rather than a *definite object* being searched for, and this is a change from the text’s situation in chapter 7. After observing the past two weeks I advise high caution and will not be giving it lecture or lab support—nor extra credit; it’s a you’re-on-your-own situation.