Preface

In an attempt to curry favor with the department chair, you're going to be helping Eli make the best use of yesterday's technology. He wants to make a CD of some of his favorite tunes (no one's told him about the iPod). He's got a two-sided CD recorder, and he wants to make a great CD for himself — chock full of his favorites. Of course, that means putting a bunch of songs on side 1, and a bunch of songs on side 2; side one holds 537Mb of data; side 2 holds only 394Mb. Eli likes a high-quality format, so that translates to exactly 11 minutes 32 seconds of music on side one, and 8 minutes and 27 seconds on side two. Eli has a collection of 200 songs to choose from. And he has rated all of them with a score between 0 and 100 (with 100 being the best).

Your mission is to help him choose the best way to pack his two-sided CD, where "best" means "with the sums of the scores of all the songs on both sides being as large as possible."

You may be thinking "Gosh . . . he'll end up with a lot of short songs!" Well . . . that *could* happen, but only if short songs are highly rated. You never know with Eli.

Details

Using yesterday's technology, Eli has carefully handwritten the song name, album title, and artist of each song with a rating next to it. Lori Agresti has typed these up for him, and Meinolf and I scanned in the type-written document. So we've got a list for you. Unfortunately, Eli failed to write down the *lengths* of the songs when he wrote them down. Fortunately, that information is available on the internet. So you'll be

- Getting the song-length information from the internet, and
- Deciding which songs to put on the CD.

The song-list is in `/u/jfh/comps/song-list.txt`; you can look at the file to determine the details of the format; it's got tab-separated fields for the artist, album name, song, and rating.

The internet information source you should use is `www.musicbrainz.org`. It's up to you to determine how to get information from that source, but your program should be general enough that it can accept a new song-file if Eli ever updates his collection — so it's no good just looking up each song by hand. To get you started, here's a kind of URL that Musicbrainz does accept:

```
http://musicbrainz.org/taglookup.html?
artist=Joni%20Mitchell&album=Miles%20of%20Aisles&
track=Love%20or%20Money
```

where I've broken the URL across three lines, which you'll need to merge to make it work. Note that the result given by this URL is a ranked list; you

should assume, for any request, that Eli's album collection contains the *first* item on the list.

Note for those who've never done any web programming: one possible way to get information from a website is "wget"—a Linux command with a great many options—but you're welcome to use any approach you like.

Constraints

The data we're giving you is good: every song-album-artist is actually in the online database, and all the rankings are integers between 0 and 100 inclusive.

Each song should be on the compilation CD *at most* once.

You must compute an optimal solution for this problem, i.e., a solution that maximizes the sum of the values of all songs that are recorded on the disk. You should implement an algorithm based on dynamic programming. Specifically, your algorithm must be based on the following recursion:

$$Q_{i,j,k} = \max(Q_{i,j,k-1}, Q_{i-w_k,j,k-1} + p_k, Q_{i,j-w_k,k-1} + p_k)$$

where in cell $Q_{i,j,k}$ you store the maximum possible profit that can be achieved if side one has i seconds of recording space, side two has space j seconds, and you can only use the first k songs for recording, and where

- w_k denotes the length of song k in seconds, and
- p_k denotes Eli's 0 - 100 score for song k.

Caching

Since there are a bunch of people taking this exam, and they'll all be making lots of requests from Musicbrainz, it's possible that Musicbrainz will get fed up and ban requests coming from cs.brown.edu. That would be bad. So you're required to implement some form of cache in which your program looks for a song first, before making yet another request from Musicbrainz. This cache should, of course, persist between invocations of the program.

Formal requirements

Your program should be submitted in a form that can be run on department Linux machines; you should submit the source code, an executable, and instructions for running the executable. The program itself should take a command-line argument which is the path to a text-file in the format of the example we gave you above; it should produce output that describes the contents of each of the two sides of the disk, together with the

- total time used, and
- the total "score",

both for each side and for the entire disk.

Your design should be well-designed, robust, handle errors gracefully, and should not crash on any input.

Your program will be run on some sample input, read by the committee, and then you'll be examined by the members of the committee, who will ask you questions about the design and functioning of your program.

Your program should be able to handle problems of size comparable to the test-data or smaller. It should fail gracefully on problems too large for it to handle.

Sources

If you need to learn about how to get data from a website, feel free to do a websearch to learn how it's done. If you need to learn about caching, feel free to consult a textbook. We ask two things:

- Please document the sources that you consult, and
- if you happen to come across a program, online, for scheduling music files on a two-sided disk, or anything close to that, you *not* read it.

Also note that if you find code to solve the exact problem that we're asking you to solve, and the code has flaws, that's *your* problem. You're probably better off writing it yourself.

Your work on the project must be your own, in the sense that you can consult written sources, but you should not talk to any other students about the project at all, nor should you use online-chat-help services, etc. Roughly speaking: it's OK to use the library or the internet equivalent; it's not OK to talk with other people or the internet equivalent. If you have any doubts about whether something is OK, please contact us before doing it.

Handing in your work

To submit your program, create a folder named `YesTech` and put ALL files of your program (your executable as well as all sources needed to compile and run your program on `goku.cs.brown.edu` — Meinolf's machine, a Linux box) into that file. Make sure that none of your files contains your name or any hints to the actual author of the program as we attempt to grade anonymously. After creating and filling the folder, go into the folder that contains YesTech and run the command `tar -czvf yestech.tar.gz YesTech` which creates the file `yestech.tar.gz`. Email this file to our comp zar Daniel Acevedo (`daf@cs.brown.edu`) who will relay your work anonymously to the graders.

You may use any publicly available resources (programming libraries, Google, published papers etc.) in order to complete this work. However, make sure that everything needed to compile and run your program is provided in the folder YesTech. Moreover, please put a file named `README` into the folder YesTech that

explains what resources you have used and where to find them. In the same file, please provide a high-level description of your approach and of the structure of your program.

Evaluation Criteria

This is a relatively easy program to write; we expect you to do it well, and expect the program to function correctly and reliably (in particular, it should respond gracefully to invalid input, and on valid input it should produce an optimal pair of song-lists). Its performance should match the best big-O performance possible with a dynamic programming approach. It should show good design, and be documented in a way that would make it maintainable by students in subsequent years. The program should be readable and well-enough written that you wouldn't mind showing it to a potential employer.

The program must perform correctly on valid input, using the methods specified (including the caching of query-results and using dynamic programming); you will not pass if this criterion is not met. The remaining criteria — good design, documentation, performance, reliability, error handling, maintainability, etc. — are all also important; an inability to handle some class or errors, or a single bad design choice ("this method really ought to be in *that* class") won't be a show-stopper, but multiple such failures can be.

Final remarks

This project is meant to be relatively easy; it's meant to give you a chance to show us that you can teach "Introductory Programming" (at whatever university might hire you) without embarrassing Brown and ruining our reputation. It's not a meant as a trial-by-fire or a race against time; there are no "trick questions" to it, and we hope that many of you will be able to complete it in a single day's work, and have plenty of time to verify and test and document at leisure.

If you have any questions, don't hesitate to ask.

Good luck!