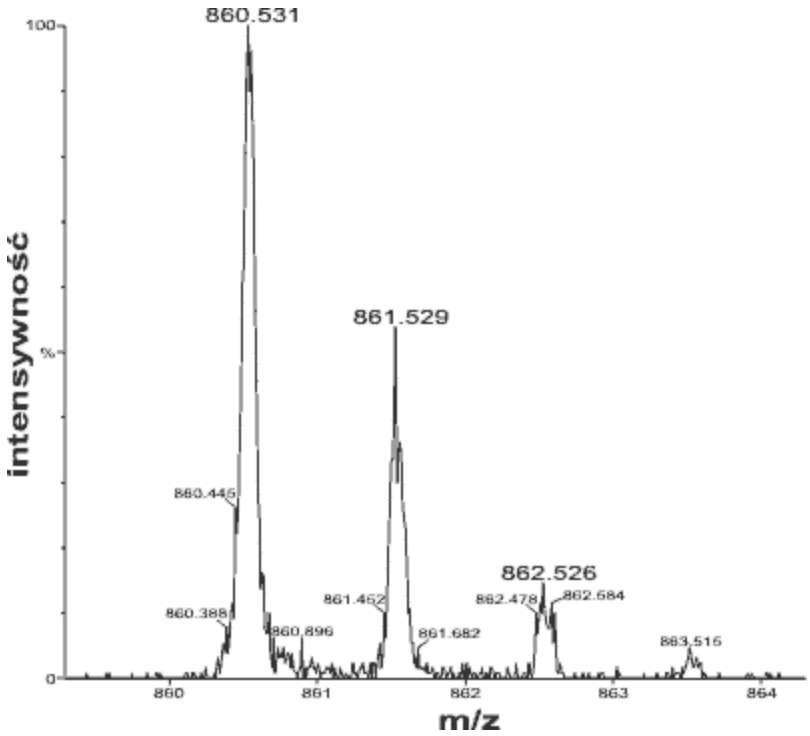
We are working on the problem of querying a mass spectrometry database. Mass spectrometry (MS) is a method of analyzing biological specimens (especially proteins). You can read about it here, but the details are not important for this task.

The input to an MS measurement is a biological sample (mix of peptides). The mass analysis output for each peptide has two parts. The first is a single “parent mass” (PM). The second is a *spectrum* of fragment ion mass to charge ratios (usually written ‘m/z’ or MZ) with its intensities. PMs usually range from 0 to 2000 and fragment masses range from 0 to 2000 each with a resolution of 0.05.
A typical raw spectrum might look like this:



As you can see, there are some clear peaks and a bunch of noise. The location of the peaks is what’s critical. Here, the *X* axis shows the MZs for each ion in the sample and *Y* axis is the corresponding intensity which quantifies the amount of the ion in relation to the amount of the most abundant ion.

The larger problem we are working on this:

Given a database of billions of spectra (DB) and single query spectrum (Q), how can we find all the spectra in the database that are “similar to” Q. For our purposes, two spectra are similar if they have peaks in many of the same places. My group is working with other researchers at UCSD to implement a system that solves this problem at a very large scale.

The system builds a specialized index data structure for the DB (INDEX) and identifies each spectrum in the database with spectrum ID (SID). The high-level steps in the algorithm we use given in this table:

Stage	**Input**	**Output**
Candidate identification and reconstruction	Q and INDEX	A list of hundred of thousands reconstructed spectra
Similarity calculation and filtering	Reconstructed spectra	A list of hundreds to thousands similar spectra.

Your task is to optimize the “candidate identification and reconstruction” portion of the algorithm.

Representing Spectra

Since most spectra have important peaks at only a small number of MZs, we will store the raw spectra using a sparse representation. We will quantize each MZ value multiplying by 20, so, e.g., 861.529 becomes the integer 17,230. We will also normalize and quantize the intensity values so they are integers between 0 and 100. For instance, we would store the spectrum shown above like this:

MZ	**Intensity**
17210	100
17230	50
17250	15
17270	5

The quantized MZ range from 0 to 40,000, so there are 40,000 possible MZs.

The Database Index

The structure of the index we use is unusual and is carefully optimized to make the candidate identification stage as fast as possible. However, it’s structure presents some challenges during candidate reconstruction.

The index is normally stored as an array of 40,000 *buckets*, one for each possible MZ. Each bucket contains a list of the SIDs that have a peak at the bucket’s MZ and the peak’s intensity.

In this project, we use only 2000 buckets instead of 40,000. This means each bucket of 2000 buckets array is mapped to 20 buckets from 40,000 original buckets.

For example, if we had this set of spectra (we've restricted ourselves to first 5 peaks of each spectrum):

SID: 0

MZ	**Intensity**
5581	2
8505	3
1003	3
10844	2
11024	29

SID: 1

MZ	**Intensity**
5203	2
5982	17
6002	3
6542	7
8563	3

SID: 2

MZ	**Intensity**
11205	4
13025	12
13035	10
13045	4
13506	6

and database index array ranging from 0 to 2000. To populate the index, for each peak of SID, we divide the MZ by 20 and put the SID in the appropriate index bucket along with the peak intensity information. For example, if we look at the first peak of **`SID 0`** which has `MZ = 5581` and `intensity = 2`, we get the bucket index by `floor(5581 / 20) = 279`, so we put **`SID 0`** with `intensity 2` at bucket `index 279`. Then, the resulting non-empty buckets of the database index would look like this (I have written them as *<SID>:<Intensity>*).

260	**279**	**299**	**300**	**327**	**425**	**428**	**542**	**551**	**560**	**651**	**652**	**675**
1:2	*0:2*	*1:17*	*1:3*	*1:7*	*0:3*	*1:3*	*0:2*	*0:29*	*2:4*	*2:10*	*2:4*	*2:6*
							0:3			*2:12*		

Spectra Reconstruction

We can now describe the candidate identification and reconstruction process. Our goal is to use the index to identify and reconstruct the candidate spectra. The candidate spectra are the spectra that share the same ion MZs with the query spectrum. This means we are only interested in where Q and the candidates have peaks in the same locations. By looking up the bucket indices that match with query spectrum peaks, we can retrieve all SIDs that have peaks at the corresponding locations. Thus, we only need to reconstruct the peaks of each candidate spectrum that match up with peaks in Q.

For example, if Q is

MZ	5162	5503	6262	6542	6622	6902	6924	6963	8344	8563	...
Intensity	3	3	3	3	4	2	5	4	5	2	...

We will `floor(Q_MZ / 20)` to get the bucket index to retrieve and we will reconstruct spectra that are residing in the retrieved buckets. There are only 2 bucket indices that match with query spectrum peaks: `floor(6542 / 20)= 327` and `floor(8563 / 20)= 428`

Index

260	**279**	**299**	**300**	**327**	**425**	**428**	**542**	**551**	**560**	**651**	**652**	**675**

1:2	*0:2*	*1:17*	*1:3*	*1:7*	*0:3*	*1:3*	*0:2*	*0:29*	*2:4*	*2:10*	*2:4*	*2:6*
							0:3			*2:12*		

We skip all other MZs that don't appear in Q and only reconstruct spectra that appear at matching bucket indices. Thus, we reconstruct the **SID 1** with its peaks appearing in buckets **327** and **428** where it matches the query peaks with **MZ = 6542** and **MZ = 8563.**

Here is the reconstructed spectrum for **SID 1** along its matching peaks with query spectrum:

SID: 1

MZ	**Intensity**
6542	7
8563	3

In the real system, we would do further processing on the reconstructed spectra, but this task, we can just print out the reconstruction JSON-style:

```
{
  1 : [[6542, 7], [8563, 3]]
}
```

The Source Code

The code we have provided in main.cpp, proceed in 5 stages:

1. It loads a database of spectra from data/804.mxs.
2. It loads a query spectra from Query5.txt.
3. It builds the index structure described above.
4. It reconstructs the spectra for each SID as described above.
5. It prints the result to 804-Query5.json.

There are also some debugging output and utility functions for printing the various data structures.

Finally, it prints out the execution time for index construction and reconstruction.

You can build the code with `make` and test it with `make test`.

The resulting binary, can be run like so:

```
main <raw_data_file> <query_file>
```
It will print the reconstructed spectra to `stdout` in JSON.

*For an initial test, we suggest you use only the first 3 spectra with the first 5 peaks from the database to understand the process. The example shown in above is the actual first 3 spectra loaded from the database, where we have restricted up to first 5 peaks for each spectrum. However, the final task should be completed using all spectra given to you by setting the `total_spectra` and `num_peaks` variables to `0`. You can also do this with 'make demo'.

Your Task

Your task is to improve the performance of reconstruct_candidates(). You can use *any* means to accomplish this subject to the following constraints:

1. You must construct the index as described above.
2. You must use the index you build to perform reconstruction (i.e,. `raw_data` is off-limits during reconstruction -- in practice, it is not available (and note that it's deleted before reconstruct_candidates() runs.))
3. Your implementation must produce the same result as `main.cpp` as determined by `check.py` (i.e., this means you need to produce equivalent JSON, but there can be white space differences).
4. Your program should take the same command line arguments as the provided code.
5. You must cite the resources you use in your solution.
6. You may not consult another person either in person or online (e.g., asking questions on Stackoverflow is not allowed, but using existing answers is fine).